Начни писать программы на С++, пользуясь этой книгой

С++ для "чайников"™

4-е издание

Для сомневающихся

Верный помощник при изучении языка С++

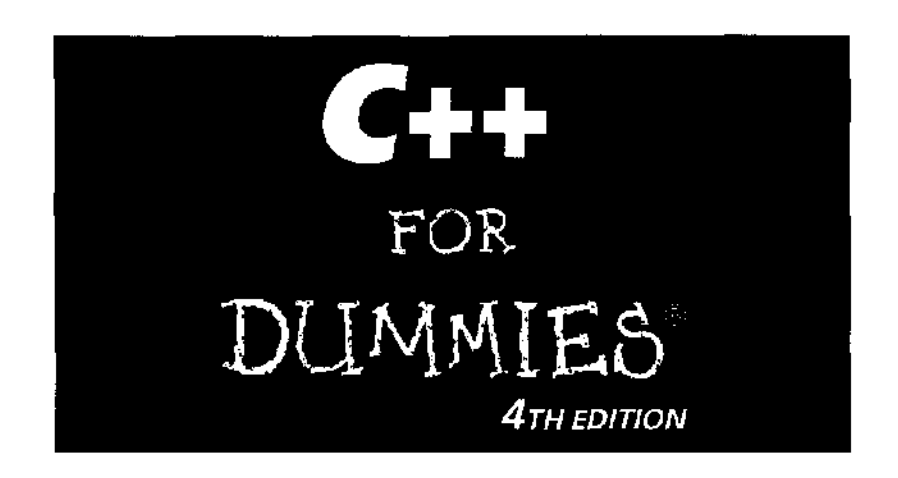


Стефан Р. Дэвис






C++
для
"ЧАЙНИКОВ"
4-е издание



by Stephen R. Davis



IDG Books Worldwide, Inc.

An International Data Group Company

Foster City, CA • Chicago, IL • Indianapolis, IN • New York, NY



Стефан Р. Дэвис



ДИАЛЕКТИКА

Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2я75
Д94
УДК 681.3.07

Компьютерное издательство ""Диалектика"

Перевод с английского *Д.М. Мищишина* и канд. техн. наук *И.В. Красикова*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info@dialektika.com, http://www.dialektika.com

Дэвис, Стефан, Р.

Д94 С++ для "чайников", 4-е издание. : Пер. с англ. : — М. : Издательский дом "Вильямс", 2003. — 336 с. : ил. : Парал. тит. англ.

ISBN 5-8459-0160-X (рус.)

Книга, которая у вас в руках, — это введение в язык программирования С++. Она начинается с азов: от читателя не требуется каких-либо знаний в области программирования. В отличие от других книг по программированию па С++, в этой книге вопрос "почему" считается не менее важным, чем вопрос "как". И поэтому перед изложением конкретных особенностей языка С++ читателю разъясняется, как они действуют в целом. Ведь каждая структурная особенность языка — это отдельный штрих единой картины. Прочитав книгу, вы сможете написать на С++ вразумительную программу и, что не менее важно, будете понимать, почему и как она работает.

Книга рассчитана на пользователей с различным уровнем подготовки.

ББК 32.973.26-018.2я75

ISBN 5-8459-0160-X (рус.)
ISBN 0-7645-0746-X (англ.)

Оглавление

Содержание

Моим друзьям и семье, которые помогли мне
стать "чайником" в еще большей степени,
чем я есть на самом деле

Об авторе

Стефан Р. Дэвис (Stephen R. Davis) — автор множества книг, включая такие бестселлеры, как C++ *для "чайников ", More C++ for Dummies* и *Windows 95 Programmingfor Dummies*. Стефан работает в компании Valtech, специализирующейся в области обучения информатике (Даллас, Техас).

Благодарности

Я считаю странным то, что на обложке любой книги, особенно такой, как эта, написано только одно имя. В действительности свой труд в создание книги *...для "чайников"* вкладывает громадное число людей. Для начала я хотел бы поблагодарить своего главного редактора Мэри Кордер (Mary Corder) и агента Клодетт Мур (Claudette Moore), направлявших меня при формировании материала этой книги. Во время работы над книгой я значительно повысил свой уровень как редактор и корректор, и в этом мне помогли редакторы первого и второго изданий Сьюзен Пинк (Susan Pink) и третьего издания Келли Юинг (Kelly Ewing) и Коллин Вильямс (Colleen Williams). Кроме того, я благодарен техническим редакторам Грегу Гантли (Greg Guntle), Гаррет Пиз (Garrett Pease) и Джеффу Банкстону (Jeff Bankston) (первое, второе и третье издания соответственно) без их участия эта книга была бы намного хуже. И если бы не помощь координатора первого и второго изданий Сьюзанны Томас (Suzanne Thomas), эта книга вообще не была бы напечатана. Однако, несмотря ни на что, на обложке представлено только одно имя, а значит, ответственность за все неточности в тексте должен нести именно его обладатель.

Хочу также поблагодарить свою жену Дженни и сына Кинси за их терпение и преданность.

И наконец, новости о последних событиях из мира животных в моем доме. Для тех, кто не читал ни одной из моих книг, объясняю, что такая сводка встречается в них регулярно.

Мои две собаки, Скутер и Труди, чувствуют себя нормально, хотя Труди почти ослеп. Наши два кролика, Бивас и Батхед, отправились на большую зеленую небесную лужайку после почти полуторалетнего проживания на газоне перед нашим домом. Мы завели двух кошек, Боба и Марли, когда писалась книга *More C++ for Dummies,* Марли умерла от кошачьей лейкемии, а Боб продолжает жить и радовать нас.

Если вы хотите пообщаться со мной по поводу программирования на C++, полуслепых собак или бродячих кроликов, пишите мне по адресу: srdavis@ACM.org.

Введение

Об этой книге

Добро пожаловать в четвертое издание книги C++ *для "чайников"*. В ней вы найдете всю необходимую для изучения C++ информацию, описанную доступным языком и не отягощенную излишними подробностями.

О чем эта книга

Книга, которая у вас в руках, — это введение в язык программирования C++.

Она начинается с азов: от читателя не требуется каких-либо знаний в области программирования (и в этом основное отличие от предыдущего издания, которое предполагает знание языка С).

В отличие от других книг по программированию на C++, в этой книге вопрос "почему" считается не менее важным, чем вопрос "как". И потому перед изложением конкретных особенностей языка C++ я старался объяснить читателю, как они действуют в целом. Ведь каждая структурная особенность языка — это отдельный штрих единой картины.

Если вы не понимаете, зачем нужны те или иные особенности языка, постарайтесь понять, как они работают. Прочитав книгу, вы сможете написать на C++ вразумительную программу и, что не менее важно, будете понимать, почему и как она работает.

Эта книга не обучает программированию для Windows. Научиться этому можно в два этапа. Сначала необходимо усвоить C++, а затем приобрести книгу *Windows 95 Programming for Dummies.*

Что такое C++

C++ представляет собой объектно-ориентированный низкоуровневый язык программирования, отвечающий стандартам ANSI и Международной организации стандартов (International Standards Organization — ISO). *Объектная ориентированность* C++ означает, что он поддерживает стиль программирования, упрощающий кодирование крупномасштабных программ и обеспечивающий их расширяемость. Будучи низкоуровневым языком, C++ может генерировать весьма эффективные высокоскоростные программы. Сертификация ANSI и ISO обеспечила переносимость C++: написанные на нем программы совместимы с большинством современных сред программирования.

Уже в самом названии содержится намек на то, что C++ является следующим поколением языка программирования С — результатом добавления новых веяний академической компьютерной мысли к старому доброму С. На C++ можно делать все то же, что и на С, и даже таким же образом. Но это нечто большее, чем просто С, наряженный в новые одежды. Дополнительные возможности C++ весьма значительны и требуют не только некоторых размышлений, по и привычки, однако результат того заслуживает.

Для опытного программиста на С язык C++ может показаться одновременно и захватывающим и расстраивающим. Представьте себе немца, читающего по-датски. Это очень похожие ситуации. Программист, использующий С, сможет понять смысл

программ на C++, но из-за значительных отличий между языками его трактовка не всегда будет адекватной. Эта книга поможет вам перейти от С к C++ настолько мягко, насколько это возможно. Однако напомним, что для читателей C++ *для "чайников"* опыт программирования на С вовсе не обязателен.

Соглашения, используемые в книге

Описываемые сообщения или любая другая информация, отображаемая на экране, будет выглядеть так:

```
Hi mom!
```

Программный код будет представлен таким же образом:

```
// программа
void main()
{
  ...
}
```

Если вы решите набирать программу вручную, следите за тем, чтобы ее текст полностью соответствовал напечатанному в книге, за исключением количества пробелов, которое может быть произвольным.

Всяческие компьютерные сообщения, такие как команды и имена функций, будут выглядеть вот так. После имен функций всегда следуют открывающая и закрывающая скобки, например `myFavoriteFunction()`.

Иногда для выполнения некоторых действий в книге рекомендуется использовать специальные команды клавиатуры. Например, когда в тексте содержится инструкция: нажать <Ctrl+C>, вам следует, удерживая нажатой клавишу <Ctrl>, нажать клавишу <C>. Вводить знак "плюс" при этом не нужно.

Время от времени будут использоваться команды меню, например File⇨Open.

В этой строке для открытия меню File и выбора нужной команды из него предлагается использовать клавиатуру или мышь.

Что можно не читать

C++ является слишком большим куском, чтобы проглотить его сразу. Вы столкнетесь и с легкими, и с достаточно сложными моментами. Чтобы уберечь вас от перенасыщения информацией, не актуальной для вас в текущий момент, некоторые технические подробности будут отмечены специальными пиктограммами (см. раздел "Использованные в этой книге пиктограммы").

Некоторые сведения общего характера будут помещаться в выделенных врезках. Почувствовав, что информация воспринимается с трудом, смело пропускайте этот раздел во время первого чтения (но помните, что по возможности его нужно будет прочитать, поскольку в конечном счете незнание каких-то моментов неизбежно скажется на ваших программах).

Нелепые предположения

Чтобы освоить материал книги C++ *для "чайников"*, совершенно не обязательно иметь какой-то опыт в программировании. Конечно, если он есть, это только плюс, но его отсутствие не должно вас тревожить.

В предыдущем издании этой книги предполагалось, что вы уже немного знакомы с языком С. Идея была в том, что изучающий С++ должен был основываться на уже имеющихся знаниях языка С. Однако такой методологический подход оказался ошибочным. Прежде всего, многие принципы С++ в корне отличаются от основополагающих принципов С, несмотря на обманчивую схожесть их синтаксиса. К тому же среди изучающих С++ большинство составляют все-таки не программисты на С, а новички в программировании.

Четвертое издание *С++ для "чайников"* начинается с основных понятий, используемых в программировании. Затем книга ведет читателя от программ из простых синтаксических конструкций до концепций объектно-ориентированного программирования. Читателю, осилившему ее всю, не составит большого труда в нужный момент произвести впечатление на друзей или блеснуть своей осведомленностью на вечеринках.

Как организована эта книга

Каждая новая структурная возможность языка будет охарактеризована следующим образом:

- ✓ что представляет собой эта возможность;
- ✓ зачем она включена в язык;
- ✓ как она работает.

Разделы книги щедро снабжены небольшими фрагментами программного кода. Каждый из них иллюстрирует представленные особенности или основные моменты некоторых моих разработок. Эти фрагменты не всегда закончены и в основном не представляют собой ничего существенного.

Примечание. Необходимость соблюдать формат книги требовала переноса очень длинных строк кода. В конце таких строк появляется стрелка, которая напоминает о том, что следует продолжать ввод, не торопясь нажимать клавишу <Enter>. Я очень старался свести эти длинные строки кода к минимуму.

В конце каждой части...

В дополнение в конце частей 2, 3 и 4 приводятся тексты серии программ BUDGET. Эти программы достаточно объемны, чтобы позволить вам получить какое-то представление о реальных программах.

К тому же очень важно понимать, как разные структурные единицы языка С++ сосуществуют в готовой программе. Поэтому, хотя забот у меня и так было предостаточно, я занялся разработкой примеров обучающих программ. Наверное, от недостатка фантазии мне пришлось потратить много времени на придумывание программных сюжетов. Хотелось бы, чтобы представленные в них особенности языка были очевидны читателю.

В конце концов в качестве примера было решено использовать программу BUDGET. Она рождается как простая, процедурно ориентированная программа. Постепенно обрастая структурными особенностями, описанными в каждой новой части, к концу книги программа BUDGET предстанет перед вами во всей красе своего объектно-ориентированного содержимого. Возможно, работа с этой программой покажется вам страшной тратой времени. Если это так, вы можете пропустить первые варианты программы (хотя, по мнению нашего редактора, замысел довольно удачный). Тем не менее я надеюсь, что, разобрав программу BUDGET, вы постигнете тайну согласованной работы возможностей С++.

Часть 1. Первоезнакомство с C++

Эта часть является отправной точкой нашего путешествия в мир C++. Вы начнете его с нелегкого испытания — написания своей первой компьютерной программы. Затем перейдете к изучению синтаксиса языка.

Часть 2. Становимся программистами

В этой части новоприобрстенные знания основных команд C++ пополнятся способностью объединять фрагменты программного кода в модули и повторно использовать их в программах.

Здесь также представлена внушающая наиболее благоговейный страх тема: указатели в C++. Если вам это ни о чем не говорит, не волнуйтесь — скоро вы обо всем узнаете.

Часть 3. "Классическое" программирование

В этой части дело запутывается все больше и больше: начинается обсуждение объектно-ориентированного программирования. По правде говоря, объектно-ориентированный подход к построению программ и есть главная причина возникновения и активного использования C++. Ведь отказавшись от объектно-ориентированных особенностей C++, мы просто возвратимся к его предшественнику — языку программирования С. В этом разделе обсуждаются такие понятия, как классы, конструкторы, деструкторы, и прочие не менее страшные термины. Не волнуйтесь, если пока что вы не совсем понимаете, о чем идет речь.

Часть 4. Наследование

Возможность наследования — это как раз то главное свойство объектно-ориентированного программирования, которое обеспечило ему известность и распространенность. Обсуждение этой одной из наиболее важных концепций, понимание которой служит ключом к эффективному программированию на C++, и является темой четвертой части. Теперь дороги назад нет: закончив освоение этого материала, вы сможете назвать себя настоящим объектно-ориентированным профаммистом.

Часть 5 Полезные особенности

К моменту знакомства с этой частью вы уже будете знать все необходимое для эффективного программирования на C++. Здесь же затрагиваются некоторые оставшиеся дополнительные вопросы. Если вы чувствуете, что голова все еще кружится от избытка с трудом воспринимаемой информации, можете пока удержаться от чтения этой части.

Часть 6. Великолепная десятка

Разве книга для "чайников" может считаться законченной без такой полезной напутствующей части? В ее единственной главе вы узнаете наилучшие способы избежать ошибок в программах.

использованные в этой книге пиктограммы



Технические подробности, которые можно пропустить при первом чтении.



Советы, которые помогут сохранить много времени и усилий.



Запомните — это важно.



Тоже важное напоминание. Это указание о том, что здесь легко допустить одну из труднонаходимых ошибок и даже не догадаться о ней.

Что дальше

Обучиться языку программирования — задача отнюдь не тривиальная. Я попытаюсь сделать это настолько мягко, насколько возможно, но вы должны будете поднатужиться и освоить некоторые элементы серьезного программирования. Так что разомните пальцы, приготовьте для книжки почетное место рядом с клавиатурой и — приступим!

Часть I

Первое знакомство с С++



"В Сети ему нет равных; благо, что есть электронная почта, потому что я не могу понять ни слова из того, что он говорит."

В этой части...

И новейшие потрясающие воображение авиационные симуляторы, и незамысловатые, но мощные вычислительные ***программы*** *состоят из одних и тех же базовых блоков. В этой части вы найдете основные сведения, необходимые для написания самых потрясающих* ***программ.***

Глава 1

Написание вашей первой программы

В этой главе...

- ✓ Постигая концепции C++
- ✓ Что такое программа
- ✓ Как пишут программы
- ✓ Создание первой программы
- ✓ Выполнение программы
- ✓ Разбор программ
- ✓ Вычисление выражений

Итак, мы на старте. Никого вокруг нет — только вы, я и книга. Сосредоточьтесь и постарайтесь овладеть некоторыми фундаментальными понятиями.

Компьютер — это поразительно быстрая, но невероятно глупая машина. Он может выполнить то и только то, что прикажешь (причем с умом!), — ни больше, ни меньше.

К нашему глубокому сожалению, компьютер не понимает привычного человеку языка — ни английского, ни русского, ни какого-либо другого. Знаю, вы хотите возразить: "Я видел компьютеры, понимающие английский". В действительности язык понимала выполняемая компьютером специально разработанная программа. (Это объяснение не совсем корректно. Но, с другой стороны, если я захочу рассказать своему маленькому сыну что-то слишком для него сложное, то постараюсь объяснить это доступными для него словами и понятиями.)

Компьютеры понимают язык, который называют машинным или языком программирования. Человеку крайне сложно разговаривать машинным языком. Поэтому в качестве посредника между компьютерами и людьми решили использовать такие языки *высокого уровни,* как C++. Они более или менее понятны людям и конвертируются в машинный язык, воспринимаемый компьютерами.

Постигая концепции C++

В начале семидесятых консорциум очень умных людей разрабатывал компьютерную систему Multix. Ее предназначением было обеспечение недорогого всеобщего доступа к графическим, текстовым и другим файлам, к электронной почте, эротике (ладно, это я уже переборщил). Конечно, это была совершенно глупая идея, и в целом проект провалился.

Небольшая группа инженеров, работающих в лабораториях Белла, решила использовать фрагменты Multix в небольшой операционной системе, которую окрестили Unix (Un-ix, Mult-ix — словом, все понятно?).

Эти инженеры не имели одной большой мощной машины, а лишь несколько маломощных машин разных производителей. Поскольку все они были разные, каждую программу требовалось перерабатывать под каждую машину. Чтобы избежать этих мучений, был разработан небольшой, но мощный язык, который назвали С.

Язык С оказался действительно мощным и очень скоро завоевал передовые позиции среди средств разработки программного обеспечения. Однако со временем в программировании появились новые технологии (например, достойное самого большого внимания объектно-ориентированное программирование), которые постепенно вытесняли язык С. Не желая остаться за бортом, инженерное сообщество усовершенствовало С, дополнив его новыми возможностями и получив в результате новый язык программирования — С++.

Язык С++ включает:

- ✓ словарь понятных для людей команд, которые конвертируются в машинный язык;
- ✓ структуру языка (или *грамматику*), которая позволяет пользователям составлять из команд работающие программы.

Примечание. Словарь известен также как семантика, а грамматика — как синтаксис языка.

Что такое программа

Программа — это текстовый файл, содержащий последовательность команд, связанных между собой по законам грамматики С++. Этот файл называют исходным текстом (возможно, потому, что он является началом всех наших страданий). Исходный файл в С++ имеет расширение .CPP, так же как файлы Microsoft Word оканчиваются на .DOC или командные файлы MS DOS имеют окончание .BAT. Расширение .CPP всего лишь соглашение, но в мире персональных компьютеров оно, по сути, стало законом.

Задача программирования — это написание такой последовательности команд, после преобразования которой в машинный язык можно получить программу, выполняющую наши желания. Такие машинно-исполнимые программы имеют расширение .EXE[1]. Процесс превращения программы С++ в исполнимую называется компиляцией.

Пока все выглядит достаточно легко, не так ли? Но это лишь цветочки. Продолжим...

Как пишут программы

Для написания программы вам нужны две вещи; редактор для создания исходного .CPP-файла и программа, которая преобразует исходный текст в понятный машине код .EXE-файла, выполняющего ваши команды. Инструмент, осуществляющий такое превращение, называется компилятором.

Современные инструменты разработки программ обычно совмещают в себе и компилятор и редактор. После ввода текста программы для создания выполнимого файла нужно только щелкнуть на кнопке.

Одна из популярнейших сред разработки — Visual С++ фирмы Microsoft. В ней можно скомпилировать и выполнить все программы, представленные в этой книге; однако не все из вас являются владельцами этого программного продукта из-за его довольно высокой стоимости даже у уличных торговцев (кстати, многие в это не поверят, но далеко не все программисты работают в Windows — есть и другие операционные системы). К счастью, существуют и общедоступные среды разработки программ С++, наиболее популярная из которых GNU С++.

---

[1] *Как правило, но, вообще говоря, это выполняется не всегда.* — Прим. ред.

Все свободно распространяющиеся программы можно найти в Internet. Некоторые из этих программ не совсем бесплатны — для их получения вы все-таки должны будете внести небольшую сумму. За использование GNU C++ вы не должны ничего платить.



GNU характеризуется шуточным определением "GNU — это не Unix" (GNU is Not Unix). Эта шутка возвращает нас к раннему периоду C++, когда признавалась только эта операционная система. На самом деле GNU представляет собой серию инструментов, разработанных представителями Фонда свободного программного обеспечения (Free Software Foundation).

GNU C++ вовсе не плод нездорового воображения группы разработчиков, а полнофункциональная среда для разработки программ на C++. Она поддерживает все структурные особенности языка C++ и может обеспечить выполнение любых про" грамм из этой книги (и изо всех других книг по C++)[2].



GNU C++ не является пакетом разработки программ для Windows. Если я разгадал ваши тайные желания, то у вас нет другого выхода, кроме как приобрести коммерческий пакет наподобие Visual C++.

Далее предполагается, что читатель посетил узел по адресу www.deloreon.com (или любой другой из сотен узлов, на которых можно найти GNU C++), загрузил последнюю версию GNU C++ и инсталлировал ее на своем компьютере, следуя приведенным в загруженном пакете инструкциям. После этого совместными усилиями будет написана наша первая программа. Задача, которую нужно реализовать, — преобразование введенной пользователем температуры по Цельсию в температуру по Фаренгейту.

Создание первой программы

Сейчас вы приступите к созданию своей первой программы на C++. Для этого потребуется ввести программный код в файл CONVERT. CPP, а потом скомпилировать его в выполнимую программу.

Введение кода

При создании любой программы на C++ первым шагом становится введение команд языка с помощью текстового редактора. Сердцем пакета GNU C++ является утилита rhide. Она представляет собой редактор, который связывает различные возможности GNU C++ в единый интегрированный пакет. В этой главе rhide будет использоваться для создания файла Convert.cpp.

1. Откройте окно MS DOS двойным щелчком на пиктограмме MS DOS в меню Windows Пуск⇨Программы.

   GNU C++ является утилитой, работающей из командной строки. Запускать rhide вы всегда будете из системного приглашения MS DOS.

2. Создайте каталог C:\CPP_For_Dummies\Chap01 (подразумевается, что рабочим диском является C).

Вы можете использовать любое угодное душе имя каталога, но намного легче манипулировать именами каталогов MS DOS, которые не содержат пробелов. Еще луч-

[2] *Пожалуй, это слишком смелое утверждение, если речь идет о книге с применением последних стандартов* C++, *но для подавляющего большинства книг данное утверждение верно.* — Прим. ред.

ше использовать имена каталогов из восьми или менее символов — тогда с ними гарантированно можно будет работать в любой среде.

Находясь в каталоге ChapOl, запустите `rhide` из приглашения MS DOS.

Создайте пустой файл, выбрав пункт New в меню File. Откроется пустое окно, в котором необходимо ввести текст программы (причем он должен точно соответствовать книжному оригиналу).



Пусть количество отступов и пробелов вас не волнует: не так важно, сколько пробелов вы поставили перед началом строки или между соседними словами. Однако C++ весьма чувствителен к другому: надо следить, чтобы все команды набирались в нижнем регистре[3].

Интерфейс rhide

Этот интерфейс существенно отличается от интерфейса Windows-программ. Программы Windows "разрисовывают" выводимую на экран информацию, и это придает им более изысканный вид.

А вот интерфейс rhide — текстовый. Он использует символы из арсенала персонального компьютера, так что облик rhide не так изящен. Интерфейс rhide не поддерживает изменение размеров окна, оно стандартного размера (80x25 символов). Но все же rhide поддерживает многое из того, к чему вы привыкли: выпадающие меню, множественные окна, интерфейс с использованием мыши, "горячие" клавиши.

Для более опытных из вас достаточно напомнить, что rhide очень похож на интерфейс набора инструментальных средств ныне "усопшего" Borland.

```
//
//  Программа для преобразования
//   градусов Цельсия в градусы Фаренгейта:
//  Fahrenheit = NCelsius  *  (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberofArgs, char* pszArgs[])
{
    // Введите температуру в градусах Цельсия
    int nNCelsius;
    cout << "Введите температуру по Цельсию:  ";
    cin  >> nNCelsius;

    // для приведенной формулы преобразования
    //вычислим преобразующий множитель
    int nNFactor;
    nNFactor = 212 - 32;

    // используем вычисленный множитель для
    // преобразования градусов Цельсия в
    // градусы Фаренгейта

    int nFahrenheit;
    nFahrenheit = nNFactor * nNCelsius/100 + 32;
```

[3] *В C++ отнюдь не запрещается использование символов в верхнем регистре — просто язык* чувствителен к регистру, *а это значит, что int main и* `Int` *Main, например, означают совершенно разные вещи.* — Прим. ред.

```
    // вывод результатов
    cout << "Температура по Фаренгейту: ";
    cout << nFahrenheit;

    return 0;
}
```

После ввода этого кода выберите Save As в меню File и сохраните файл под именем `Conversion.cpp`.

Хотя это вам может показаться и не очень впечатляющим, но только что вы создали вашу первую программу на C++!

Создание выполнимого файла

После сохранения на диске исходного файла `Conversion.cpp` самое время сгенерировать выполнимый машинный код.

Для этого нужно выбрать пункт Make меню Compile или просто нажать клавишу <F9>. Интерфейс `rhide` откроет в нижней части экрана еще одно небольшое окно, отражающее ход процесса компиляции. Если все в порядке, после сообщения `Creating Conversion.exe` вы увидите слова `no errors`.

Ошибки инсталляции GNU C++

В процессе инсталляции могут возникнуть довольно распространенные ошибки, которые испортят ваши выдающиеся достижения в программировании. Две наиболее распространенные ошибки можно диагностировать, пытаясь скомпилировать программу.

Сообщение `Bad command or file name` означает, что MS DOS не может найти `gcc.exe`, т.е. компилятор GNU C++. Причиной этого может быть неправильная инсталляция продукта или некорректное задание пути `c:\djgpp\bin` к каталогу, где находится `gcc.exe`. Попытайтесь реинсталлировать GNU C++ и проверьте, присутствует ли в файле `autoexec.bat` строка `SET PATH=c:\djgpp\bin;%PATH%`. Перезагрузите компьютер.

Сообщение `gcc.exe: Conversion.cpp: No such file or directory (ENOENT)` означает, что `gcc` не понимает используемых длинных имен файлов (что характерно для MS DOS). Эту проблему можно решить, подредактировав файл `c:\djgpp\djgpp.env` и присвоив в нем свойству `LFN` значение Y.

И еще одно предупреждение: GNU C++ не понимает имен файлов, которые содержат пробелы. В этом случае не поможет даже флажок, разрешающий длинные имена.

Когда компилятор GNU C++ сталкивается в программе с какой-либо некорректностью, он генерирует сообщение об ошибке. Ошибки в программном коде являются таким же распространенным явлением, как снег на Аляске. Вы обязательно столкнетесь с многочисленными предупреждениями и сообщениями об ошибках, возможно, даже при работе с простенькой программой `Conversion.cpp`. Чтобы продемонстрировать процесс исправления ошибок, изменим в строке 13 `cin>>nCelsius;` на `cin>>>nCelsius;`.

Это нарушение кажется совсем невинным — и вы, и я вскоре бы о нем забыли. Но при компиляции `rhide` генерирует следующие сообщения:

```
Compiling: Conversion.cpp
In function 'int main(int, char **)':
Conversion.cpp(13) Error: parse error before '>'
There were some errors
```

Таким многословным образом компилятор сообщает, что GNU C++ не может понять значения записи “>>>” из 13-й строки программы.

Термин *parse* обозначает, что ошибка была найдена при проведении синтаксического анализа команд C++.

Отредактируйте файл, удалив лишний знак ">". Теперь, нажав <F9>, вы благополучно создадите выполнимую программу `Conversion.exe`.

Почему C++ так требователен

Как видим, C++ смог определить, что мы испортили в предыдущем примере. Однако если **GNU** C++ нашел ошибку, то почему же он сам не решит эту проблему — и дело с концом?

Ответ достаточно прост. Хотя в данном случае GNU C++ считает, что мы всего лишь допустили опечатку при вводе символов “>>”, полностью положиться на его интуицию нельзя. Ведь правильной командой в действительности может оказаться совершенно другая, не имеющая никакого отношения к ошибочной команде. Если бы компилятор исправлял ошибки так, как считает нужным, то **GNU** C++ скрывал бы от разработчиков многие реальные проблемы.

Требуется много усилий и времени, чтобы найти скрытую в программе ошибку. Намного лучше позволить найти эту ошибку компилятору. Мы тратим наше драгоценное время, создавая ошибки. Но зачем же расходовать его еще и на их поиск, если **GNU** C++ может их выловить, не тратя нашего времени. Каким, как вы думаете, будет мой выбор?..

Выполнение программы

Пришло время испытания вашего нового творения. Для выполнения программы нужно запустить файл CONVERT.EXE И обеспечить его входными данными. Полученный результат можно использовать для анализа.

Чтобы запустить программу из среды GNU C++, нужно выбрать пункт Run из меню Run или нажать <Ctrl+F9>.

При этом откроется окно, в котором вам предложат ввести температуру по Цельсию. Для проверки правильности внесите какую-то заранее известную температуру, например 100°. После нажатия клавиши <Enter> программа возвращает эквивалентную температуру по Фаренгейту, т.е. 212°. Однако, так как rhide закрывает окно сразу же по завершении программы, увидеть результат вы не сможете; rhide открывает предупредительное окно с сообщением, что программа завершена с нулевым кодом ошибки. Несмотря на устрашающее выражение "код ошибки", ноль как раз обозначает отсутствие ошибок в процессе выполнения программы.

Чтобы увидеть результат выполнения программы, щелкните на пункте User Screen меню Windows или нажмите <Alt+5>. Это окно отображает предыдущее окно сеанса MS DOS. В нем можно увидеть последние 25 строк выполнения программы, в том числе вычисленное значение температуры по Фаренгейту.

Поздравляю! Используя GNU C++, вы только что ввели, скомпилировали и запустили свою первую программу.

GNU - это не Windows

Заметьте, что пакет GNU C++ не предназначен для разработки программ Windows. Написать Windows-приложение с помощью GNU C++ теоретически можно, но без использования внешних библиотек, как в Visual C++, сделать это весьма непросто.

Windows-программы имеют ярко выраженный визуально-ориентированный оконный интерфейс. А Conversion.exe является 32-битовой программой, которая выполняется в среде Windows, но Windows-программой в настоящем смысле ее не назовешь.

Если вы не знаете, чем 32-битовая программа отличается от 16-битовых, не волнуйтесь об этом. Как уже отмечалось, эта книга не о написании программ для Windows. Интерфейс программ, разработанных нами в среде GNU C++, представляет собой набор командных строк в окне MS DOS.

Начинающим Windows-программистам огорчаться не следует: ваше время не пропадет зря. Изучение C++ совершенно необходимо как предварительное условие для написания Windows-программ.

Помощь в GNU C++

GNU C++ обеспечивает разработчиков системой помощи через пользовательский интерфейс rhide. Разместите свой курсор на конструкции, смысл которой неясен, и нажмите <F1>. Раскроется окно с имеющейся по этому вопросу информацией. Для отображения тематического списка помощи нужно выбрать Help⇨Index.



Справочная информация, предоставляемая GNU C++, не так исчерпывающа, как справка некоторых других инструментальных средств (например, Microsoft Visual C++). Если разместить курсор на слове "int" и нажать <F!>, появится окно, описывающее редактор, — но ведь это не совсем то, что мы искали. Внимание GNU C++ в основном сосредоточено на библиотеке функций и опциях компиляции. К счастью, после того как вы овладеете языком C++, помощи GNU C++ окажется достаточно для разработки большинства приложений.

Разбор программ

Хотя разбор программы, написанной другим, — вещь не самая впечатляющая, но на этапе вашего становления как программиста заниматься этим очень даже полезно. Рассмотрим далее программу Conversion.cpp и найдем элементы, общие для всех программ.

Определение структуры программ C++

Каждая программа, написанная с использованием материала этой книги, в своей основе будет иметь одну и ту же базовую схему:

```
//Это комментарии, которые компьютер игнорирует
#include <stdio.h>
#include<iostream.h>
int main(int nNumberofArgs, char* pzArgs[ ])
{
    //...здесь записывается код программы...
    return 0;
}
```

Если не вникать в детали, то выполнение программы начинается с кода, который помещен между открывающей и закрывающей скобками.

Использование в исходном коде комментариев

Нетрудно заметить, что первые несколько строк Conversion.cpp являются обычным текстом. Значит, или GNU C++ оказался более понятливым, чем я его представил, или — что вероятнее всего — этот код предназначается для человеческих глаз.

Оформленные таким образом строки называют комментариями. Чаше всего в комментариях программист объясняет конкретные действия, которые он собирается реализовать в следующем фрагменте кода. Компилятор комментарии игнорирует.

Комментарии в С++ начинаются с двойной косой черты (//) и заканчиваются переходом на новую строку. В их тексте можно использовать любые символы. Длина комментариев не ограничена, но, так как желательно, чтобы они не превосходили размеров экрана, обычно придерживаются нормы не более 80 символов.

Во времена печатных машинок перевод каретки означал начало новой строки. Но ввод с клавиатуры — это не печатание на машинке. В этом случае новая строка является символом, который завершает текущую командную строку.

Допустима и другая форма комментариев, при которой игнорируется все, что /* заключается в такие скобки */; однако эта форма комментариев в С++ почти не используется.

Присутствие в программах игнорируемых компьютером команд С++ (или любого другого языка программирования) может покачаться странным. Однако все компьютерные языки предлагают те или иные способы оформления комментариев. Объяснения программиста раскрывают ход его мыслей при написании программного кода. Ведь замыслы программиста могут быть совсем неочевидными для людей, которые захотят воспользоваться программой или ее модифицировать. Да и сам автор программы, взглянув на нее через месяц, не всегда сможет вспомнить ее суть.

Использование инструкций в программах

Все программы С++ в своей основе имеют то, что называют инструкциями. В этом разделе рассмотрим такие из них, которые составляют остов программы Convert.

Инструкция — это команда, которую понимает компилятор. Все инструкции, кроме комментариев, оканчиваются точкой с запятой (для комментариев на то есть свои причины, но все же иногда это неудобно; мне кажется, что во избежание путаницы после комментариев точку с запятой следовало бы ставить тоже).

При запуске программы первой выполняется инструкция, находящаяся после открывающей фигурной скобки, а затем поочередно выполняются и все остальные инструкции.

Просмотрев программу, можно увидеть, что пробелы, символы табуляции и новой строки появляются на протяжении всей программы. Переход на новую строку осуществляется практически после каждой инструкции. Все эти символы называют *непечатаемыми*, так как на экране монитора их увидеть нельзя.



Для повышения удобочитаемости допускается добавление символов пробела в любом месте программы (но не внутри слов!).

Игнорируя пропуски, язык С++ учитывает регистр. Например, переменные fullspeed и FullSpeed, с его точки зрения, не имеют между собой ничего общего.

Объявления

Строка int nNCelsius; является инструкцией объявления. *Объявление*— это инструкция, которая определяет переменную. Переменная — это контейнер, в котором хранятся значения некоторого типа. Переменная может содержать числовые или символьные значения.

Термин "переменная" был заимствован из алгебры, где он является стереотипным для следующих выражений:

```
x = 10
У = 3 * x
```

Во втором выражении y — множество, которое задается формулой 3*x. Но что такое x? Переменная x играет роль контейнера для хранения каких-то значений. В нашем случае значением x является 10, но с таким же успехом можно определить значение x равным 20, 30 или -1. Вторая формула имеет смысл при любом числовом значении x.

В алгебре можно начать работу с выражений типа x = 10. Программируя на С++, переменную x также необходимо определить перед ее использованием.

В С++ переменная имеет тип и имя. Переменная, определенная в строке 11, называется nNCelsius. Согласно объявлению она целочисленная (подобные названия типов, наверное, имеют целью развить у программистов ассоциативное мышление — тип int представляет собой сокращенное integer).

Для С++ имя переменной не имеет никакого специфического значения. Имя должно начинаться с букв английского алфавита A-z или a-z[4]. Остальные символы могут быть буквами, цифрами от 0 до 9 или подчеркивающей чертой (_). Имена переменных могут быть настолько длинными, насколько это вам удобно.



Существует негласная договоренность о том, что имена переменных должны начинаться *со* строчной буквы. Каждое слово внутри имени переменной пишется с прописной буквы, например myVariable.



Старайтесь давать переменным короткие, но наглядные имена. Избегайте таких имен, как x, потому что они не несут никакого смысла. Примером достаточно наглядного имени переменной может служить lengthOfLineSegment.

Генерирование вывода

Строки, начинающиеся с cout и cin, называют инструкциями ввода-вывода, или сокращенно I/O (input/output) (как и все инженеры, программисты любят сокращения и аббревиатуры).

Первая инструкция I/O выводит фразу "*Введите температуру по Цельсию*" в cout (произносится как "си-аут"). В С++ cout — это имя стандартного устройства вывода. В нашем случае таким устройством является монитор.

В следующей строке все происходит с точностью до наоборот. Со стандартного устройства ввода мы получаем значение и сохраняем его в целой переменной nNCelsius. Стандартным устройством ввода для С++ в данном случае служит клавиатура. Этот процесс является аналогом упоминаемой выше алгебраической формулы x = 10 в С++. Профамма будет считать значением nNCelsius любое целое число, введенное пользователем.

Вычисление выражений

Почти все профаммы выполняют вычисления того или иного вида. В С++ выражением называется инструкция, которая выполняет какие-либо вычисления. Иными словами, выражение — это инструкция, которая имеет значение. Команда, генерирующая это значение, называется оператором.

Например, в программе Conversion можно назвать "вычисляющим выражением" совокупность строк с объявлением переменной nNFactor и определением ее значения

---

[4] *Вообще говоря, имя может также начинаться с символа подчеркивания, хотя на практике это используется довольно редко.* — Прим. ред.

как результата вычислений. Эта команда вычисляет разницу между 212 и 32. В данном примере оператором является знак "минус" ("-"), а выражением — "212-32".

Сохранение результатов выражения

Разговорный язык может быть далеко не однозначным. Яркий тому пример — слово *равный*. Оно может употребляться в значении "одинаковый" (например, *равные силы),* а может применяться в математике для построения выражений типа "у равен утроенному х".

Чтобы избежать двусмысленности, программисты на С++ называют знак "=" *оператором присвоения*. Оператор присвоения сохраняет результат выражения, находящегося справа от "=", в переменной, записанной слева. Программисты говорят, что "переменной nNFactor присвоено значение 212-32".

Обзор программы Convert продолжается...

Второе выражение, представленное в Conversion.cpp, несколько сложнее первого. В нем используются всем известные математические символы: "*" для умножения, "/" для деления, "+" для сложения. В этом случае, однако, вычисления выполняются не просто с константами, а с переменными.

Значение переменной nNFactor (кстати, уже вычисленное) умножается на значение переменной nNCelsius (которое было введено с клавиатуры). Результат делится на 100 и к нему прибавляется 32. Результат всего выражения приводится к целому типу и присваивается переменной nFahrenheit.

Последние команды выводят строку *"Температура по Фаренгейту:"* и отображают значение переменной nFahrenheit.

Глава 2

Премудрости объявления переменных

В этой главе...

- ✓ Объявление переменных
- ✓ Объявление разных типов переменных
- ✓ Объявления типов переменных
- ✓ Выражения смешанного типа

Одним из основных в C++ является понятие *переменной*. Переменную можно представить, как небольшую шкатулку, в которой хранятся вещи для дальнейшего многократного использования. Понятие переменной заимствовано из математики. Инструкция вида

```
x = 1
```

сохраняет значение 1 в переменной x. После такого присвоения математики могут использовать переменную x вместо константы 1, пока не изменят значение x на другое.

В C++ переменные используются таким же образом. После присвоения x = 1; и до следующего изменения содержимого переменная x становится обозначением числа 1 в программе. При этом говорят, что значение x есть 3.

К сожалению, в C++ возни с переменными несколько больше, чем в математике. Эта глава как раз и повествует о заботах, связанных с использованием переменных в C++.

Объявление переменных

Все числа, с которыми работает C++, хранятся в небольших "ларцах", которые называются переменными. В математике с переменными обращаются достаточно свободно. Допускаются формулировки наподобие

$$\begin{cases} (x+2) = y/2 \\ x + 4 = y \end{cases}$$, найти ? и v.

Уверен, вам не нужно объяснять, что такой способ задания переменных действительно однозначен. К сожалению, C++ не так сообразителен (как я уже упоминал, компьютеры ну просто очень глупы!).

Прежде чем использовать в программе новую переменную, вы должны ее объявить:

```
int x;
x = 10;
int y;
y = 5;
```

Таким образом, мы объявили переменные x, y и определили, что они могут содержать значения типа int (типы переменных обсуждаются в следующем разделе). Объявлять переменные можно в любом удобном для вас месте программы, но обязательно перед их использованием.

Объявление разных типов переменных

Вы, вероятно, думаете, что переменная в математике — это совершенно аморфное хранилище для любой информации, которая взбредет в голову. Ведь в принципе можно свободно написать следующее:

```
x = 1;
x = 2.3
x = "Это - предложение."
x = Texac
```

Но С++ не настолько гибкий язык. (С другой стороны, С++ очень легко может справиться с совершенно непосильными для нас задачами. Например, ему ничего не стоит сложить миллион чисел всего за одну секунду[5].) В С++ переменные могут хранить значения только одного типа. Причиной тому является большая разница в размерах памяти, необходимой для хранения значений переменных разных типов. Если некоторые данные программы могут состоять всего из одного числа, то довольно часто разработчикам приходится манипулировать целыми предложениями.

Добавлю, что особенности использования переменных разных типов различны. Пока вы встречались только с переменными типа int:

```
int x;
x = 1;
```

В С++ тип int определяет множество целых чисел. Напомню, что целым называется число, не имеющее дробной части.

Целые числа используют для самых разных видов вычислений. Детально этому учат в младшей школе, приблизительно до шестого класса, и лишь потом начинается путаница с дробями[6]. Та же тенденция характерна и для С++, в котором более 90% всех переменных имеют тип int[7].

К сожалению, иногда использование в программах переменных типа int приводит к ошибочным результатам. Когда в первой главе вы работали с программой, преобразующей температуру, существовала (пусть неявно) проблема: программа могла работать только с целыми значениями температуры. Отмечу, что в этой конкретной программе использование исключительно целых чисел вряд ли приведет к отрицательным последствиям. Но при проведении серьезных метеорологических исследований усечение дробной части температурных значений может поставить под вопрос истинность полученных результатов. Простейшим примером может служить определение значений температуры для книги рекордов Гиннеса. В этом случае требуется высокая точность вычислений, но из-за отбрасывания дробных частей достичь ее невозможно.

Ограничения, налагаемые на целые числа в С++

Целочисленные переменные в С++ представляются типом int. На переменные этого типа накладываются те же ограничения, что и на их эквиваленты в математике.

---

[5] *Заметим, что складывает числа не язык, а компьютер — язык только передает ему задание.* — Прим. ред.

[6] *Автор имеет в виду американскую школу; впрочем, современные тенденции в отечественной школе примерно те же.* — Прим. ред.

[7] *Эта величина опять-таки существенно зависит от типа разрабатываемой программы.* — Прим. ред.

Округление до целых значений

Рассмотрим проблему вычисления среднего трех чисел. Введем три целочисленные переменные — nValue1, nValue2, nValue3. Среднее значение вычисляется по формуле

```
(nValue1+ nValue2 + nValue3) / 3
```

Поскольку все три значения являются целыми, их сумма тоже будет целым числом. Например, сумма чисел 1, 2 и 2 равна 5. Но если 5 поделить на 3, получим $1\frac{2}{3}$, или 1,666.... В отличие от людей (обладающих разумом), компьютеры (которым он свойственен далеко не всегда) приводят полученный результат к целому значению, просто отбрасывая его дробную часть. При этом 1,666 утратит свой "дьявольский" остаток и превратится в 1.

Для многих приложений усечение дробной части числа не представляет большой проблемы. Зачастую оно может быть даже полезным (разумеется, сказанное не касается математических или экономических программ). Однако такое округление целых может весьма пагубно сказаться на работе других программ. Рассмотрим следующую, эквивалентную приведенной выше формулу:

```
nValue1/3 + nValue2/3 + nValue3/3
```

Подставляя в нее те же значения 1, 2 и 2, в результате получим 0. Это случилось потому, что каждое слагаемое оказалось числом, меньшим 1. Компьютер округлил их до 0, а сумма трех нулей, как известно, равна 0. Так что такого приведения к целочисленным значениям, вообще говоря, нужно избегать.

Ограничения диапазона

Второй проблемой переменной типа int является ограниченный диапазон возможных ее значений. Максимальным значением обычной целочисленной переменной является число 2 147 483 647, минимальным — -2 147 483 648, т.е. общий диапазон — около 4 млрд чисел[8].

Решение проблемы усечения дробной части

Рассмотренные особенности переменных типа int делают невозможным их использование в некоторых приложениях. Но, к счастью, С++ умеет работать и с десятичными числами, которые могут иметь ненулевую дробную часть (математики называют их *действительными числами).* Используя действительные числа, можно избежать большинства перечисленных проблем. Заметьте, что десятичные числа могут иметь ненулевую дробную часть, а могут и не иметь, оставаясь действительными. В С++ число 1.0 является таким же действительным числом, как и 1.5. Эквивалентным им целым числом является просто 1.

В С++ действительные числа определены как числа с плавающей точкой, или просто float. Используя выражение "с плавающей точкой", имеют в виду, что десятичную запятую (или используемую вместо нее в программах точку) в десятичных числах можно перемешать вперед и назад настолько, насколько этого требуют вычисления. Действительные переменные объявляются так же, как и переменные типа int:

```
float fValue;
```

---

[8] *Вообще говоря, диапазон представимых типом int значений определяется множеством факторов — в первую очередь компилятором, на выбор типа int которого оказывает огромное влияние тип компьютера, поэтому считать определенным раз и навсегда, что диапазон значений int простирается от* -2^{32} *до* $+2^{32}-1$, *нельзя.* — Прим. ред.

Начиная с этой строки, во всей остальной части программы переменная fValue может принимать значения типа float. Тип уже объявленной переменной изменить нельзя: fValue является действительной переменной и останется ею до конца программы. Рассмотрим, как решается присущая целочисленным переменным проблема отбрасывания дробной части. Для этого в объявлении все переменные определим как действительные (тип float):

1/3 + 2/3 + 2/3

Это эквивалентно выражению

0.333... + 0.666... + 0.666...,

которое равно

1.666...

Ограничения, налагаемые на числа с плавающей точкой

Хотя числа с плавающей точкой могут решить многие вычислительные проблемы, на их использование тоже существуют ограничения. Проблемы отчасти противоположны тем, которые характерны для целочисленных переменных. Действительные переменные не могут использоваться для перечисления, с ними сложнее работать компьютеру, и они тоже страдают от ошибок округления (хотя намного меньше, чем переменные типа int).

Перечисление

Использовать переменные с плавающей точкой для простого перечисления нельзя. C++ не умеет определять, какое целочисленное значение подразумевается под действительным числом.

Например, ясно, что 1.0 есть 1. Но что такое 0.9 или 1.1? Следует ли их рассматривать как 1? Так что C++ избегает многих проблем, требуя использовать при перечислении только целые значения.

Скорость вычислений

Исторически сложилось так, что процессор компьютера выполняет операции с целыми числами гораздо быстрее, чем с действительными. Для сложения 1000 целых чисел процессору может потребоваться столько же времени, сколько для выполнения только 200 вычислений с плавающей точкой.

Однако с увеличением производительности микропроцессоров проблема скорости вычислений становится все менее важной. Большинство современных процессоров содержат специальные вычислительные схемы, которые позволяют вычислять выражения с плавающей точкой почти так же быстро, как и целочисленные выражения.

Потеря точности

Действительные переменные не могут решить всех вычислительных проблем. Обычно их точность ограничена приблизительно шестью разрядами, но есть и расширенный варианте типа для действительных чисел, который может содержать после десятичной точки до 15 значимых разрядов.

Чтобы понять эту проблему, представим 1/3 в виде бесконечной последовательности 0.333.... Однако математическое понятие периода в программировании не имеет смысла, так как точность компьютерных вычислений ограничена и где-то наша дробь должна оборваться (что зависит от использованного для хранения числа типа переменной). Поэтому, усреднив числа 1, 2, 2, мы получим не точное, а приблизительное значение 1.666667.

В некоторых случаях ошибки округления может исправлять сам С++; например, выводя информацию на экран, вместо числа 0.999999 С++ выдаст пользователю значение 1.

Ограниченность диапазона

Тип данных float также ограничен, хотя диапазон чисел с плавающей точкой намного обширнее диапазона целочисленных переменных. Максимальным значением типа int является число чуть больше 2 млрд. Максимальное значение переменной типа float приблизительно равно Ю-[18], т.е. 1 с 38 нулями[9].



Представляя переменные с плавающей точкой в стандартном виде, С++ учитывает после десятичной точки только первые шесть разрядов. Остальные разряды становятся жертвами ошибочных округлений. Поэтому действительная переменная может хранить значение 123 000 000 без потери точности из-за округления, в то время как значение 123 456 789 приведет к ошибке округления.

Объявления типов переменных

Вы уже знаете, что все переменные в программе должны быть объявлены и что им должен быть назначен тип. В табл. 2.1 представлен список некоторых стандартных типов переменных языка С++ с указанием их достоинств и недостатков.

*Таблица **2.1.** Переменные С++*

ПЕРЕМЕННАЯ	ПРИМЕР	ХАРАКТЕРИСТИКА
int	1	Простые положительные или отрицательные числа, используемые для перечисления
float	1.0F	Действительные числа
double	1.0	Расширенная версия float: использует больше памяти, допускает работу с большим диапазоном и обеспечивает более высокую точность вычислений
char	c	Символьный тип; значением переменных может быть символ алфавита, цифра, знак препинания или знак арифметической операции. Не годится для арифметических операций
string	"this is a string"	Строка символов, составляющая предложение
long	10L	Потенциально расширенная версия типа int. В GNU С++ и Microsoft Visual С++ разницы между типами long и int нет

Следующий оператор объявляет переменные lVariable **типа** long **и** dVariable **типа** double **и присваивает им начальные значения:**

```
//объявление переменной и установка ее равной 1
long lVariable;
lVariable = 1
```

[9] *Следует отдавать себе отчет, что это не означает, будто тип float может представить 10^{38}* разных *значений; вспомните, что говорилось несколько выше о количестве разрядов в числах этого типа.* — Прим. ред.

```
//объявление переменной типа double и ее инициализация
double dVariable;
dVariable = 1.0;
```



Объявить и инициализировать переменную можно одним оператором:

```
int nVariable = 1; //объявление переменной и ее инициализация
```

Единственное преимущество инициализации переменной в объявлении — уменьшение размеров текстов программ. Но даже и это весьма важное достижение.

Переменная типа char может содержать единственный символ, в то время как строковая переменная — строку символов. Поэтому *a* можно интерпретировать и как символ *a*, и как строку, содержащую только букву *a* (в действительности string не является типом переменной, но в большинстве случаев его можно рассматривать именно так). В главе 9, "Второе знакомство с указателями", вы найдете детальное описание этого типа данных.



Символ *a* и строка *a* — это далеко не одно и то же. Если вы захотите присвоить символьной переменной строковое значение (или наоборот), вы не сможете этого сделать даже в том случае, когда строка содержит единственный символ.

Типы констант

Константой называют произвольную постоянную величину (например, 1, 0.5 или 'c'). Подобно переменным, константы имеют свой тип. В выражении n = 1; константа 1 имеет тип int. Чтобы привести 1 к типу long, нужно написать n = 1L;. Для лучшего понимания можно провести следующую аналогию: если под 1 понимать поездку на грузовике, то 1L можно интерпретировать как путешествие на лимузине. Их маршруты могут быть совершенно одинаковыми, но согласитесь, что путешествовать вторым способом гораздо удобнее.

Константу 1 можно также привести к действительному числу 1.0. Однако заметим, что по умолчанию типом действительной константы является double. Поэтому 1.0 будет числом типа double, а не float.

Специальные символы

Для работы с любыми печатаемыми символами можно использовать переменные типа char или string. Но значениями переменных, используемых в качестве символьных констант, могут быть и непечатаемые символы. В табл. 2.2 приведено описание некоторых важных непечатаемых символов.

Таблица 2.2. Специальные символы

Символьные константы	Обозначение действий
\n	Новая строка
\t	Табуляция
\0	Нулевой символ
\\	Обратная косая черта

С символом новой строки вы уже встречались раньше. Он позволяет разделить строку в любом месте на две части. Например, строка

```
"Это первая строка\nЭто вторая строка"
```

при выводе на экран будет выглядеть так:

```
Это первая строка
Это вторая строка
```

По аналогии символ табуляции \t перемещает выводимую информацию к следующей позиции табуляции. В зависимости от типа компьютера, на котором вы запустите программу, эта позиция может изменяться. Символ "обратная косая черта" используется для обозначения специальных символов, поэтому, чтобы вывести его на экран, необходимо записать два символа: \\.

Коллизии между C++ и именами файлов MS DOS

В MS DOS для разделения имен файлов в указаниях пути используется символ обратной косой черты. Так, root\folderA\file представляет собой путь к файлу file в папке folderA, которая является подкаталогом каталога root.

К сожалению, функциональное предназначение обратной косой черты в MS DOS и C++ не совпадает. Обратная косая черта в C++ используется для обозначения управляющих символов, а ее саму можно вывести с помощью символов \\. Поэтому путь MS DOS root\folderA\file представляется в C++ строкой root\\folderA\\file.

Выражения смешанного типа

C++ позволяет использовать в одном выражении переменные разных типов. Например, позволено складывать целые и действительные переменные. В следующем выражении переменная nValue является целой:

```
//в следующем выражении перед выполнением операции сложения
// значение nValuel преобразуется к типу double
int nValuel = 1;
double fValue = nValuel + 1.0;
```

Выражение, в котором два операнда относятся к разным типам, называется *выражением смешанного типа.* Тип генерируемого в результате значения будет соответствовать более мощному типу операнда. В нашем случае перед началом вычислительного процесса nValuel конвертируется в тип double. По тому же принципу выражение одного типа может быть присвоено переменной другого типа, например:

```
//в следующем задании целая часть
// fVariable сохраняется в nVariable
float fVariable = 1.0;
int nVariable;
nVariable = fVariable;
```



Если переменная в левой стороне равенства относится к типу менее мощному, чем переменная справа, то при таком присвоении можно потерять точность или диапазон значений (например, если значение переменной fVariable превышает диапазон допустимых значений переменной nVariable).

Соглашения по именованию

Вы могли заметить, что имя каждой переменной начинается с определенного символа, который, как может показаться, совсем ни к чему (эти специальные символы приведены в таблице ниже). С помощью соглашений по использованию этих символов можно мгновенно распознать, что dVariable — это переменная типа double.

Эти символы помогают программисту распознавать типы переменных, не обращаясь к их объявлениям в другом месте программы. Так, нетрудно определить, что в представленном ниже выражении осуществляется присвоение смешанного типа (переменная типа long присваивается целочисленной переменной):

```
nVariable = lVariable;
```

- Для C++ использование этих специальных символов не имеет никакого значения. При желании для обозначения переменной типа int вы можете использовать любую другую букву. Но использование "первобуквенного" соглашения упрощает понимание, и многие программисты постоянно используют подобные схемы в своей работе.

Символ	Тип
n	int
l	long
f	float
d	double
c	character
sz	string

Преобразование типа большего размера в меньший называется *понижающим приведением* (demotion), а обратное преобразование — *повышающим приведением* (promotion).



Использование в C++ выражений смешанного типа — идея далеко не самая блестящая; их лучше избегать, не позволяя C++ делать преобразования за вас.

Глава 3

Выполнение математических операций

В этой главе...

- ✓ Бинарная арифметика
- ✓ Анализ выражений
- ✓ Определение порядка операций
- ✓ Выполнение унарных операций
- ✓ Использование операторов присвоения

Переменные придуманы математиками не только для того, чтобы было что описывать и в чем сохранять какие-то значения. Над переменными можно выполнять самые разные действия: складывать, перемножать, вычитать и т.д. Список возможных операций достаточно обширен.

Эти основные математические операции используются и в программах C++. Ведь приложения без вычислительных возможностей себя совершенно не оправдывают. Кому нужна страховая программа, которая не может подсчитать даже суммы взносов?

Операции C++ внешне идентичны обыкновенным арифметическим операциям, выполняемым на клочке бумаги; разве что применяемые в вычислениях переменные перед использованием нужно объявлять:

```
int var1;
int var2 = 1;
var1 = 2 * var2;
```

В этом примере объявлены две переменные, var1 и var2. Переменной var2 присвоено начальное значение 1, var1 определена как результат удвоения переменной var2.

В этой главе вы найдете описание всего множества математических операторов C++.

Бинарная арифметика

Бинарными называются операторы, которые имеют два аргумента. В выражениях типа var1 *op* var2 оператор *op* бинарный. Самыми распространенными бинарными операторами являются простые математические операции, изучаемые еще за школьными партами. Бинарные операции, которые поддерживает C++, приведены в табл. 3.1.

Таблица 3.1. Математические операции в порядке приоритета

ПРИОРИТЕТ	ОПЕРАТОР	ЗНАЧЕНИЕ
1	+ (унарный)	Реально ничего не изменяет
1	- (унарный)	Возвращает противоположное по знаку, равное по модулю значение
2	++ (унарный)	Оператор инкремента, увеличивает значение аргумента на 1
2	— (унарный)	Оператор декремента, уменьшает значение аргумента на 1

ПРИОРИТЕТ	ОПЕРАТОР	ЗНАЧЕНИЕ
3	* (бинарный)	Умножение
3	/ (бинарный)	Деление
3	% (бинарный)	Остаток (деление по модулю)
4	+(бинарный)	Сложение
4	- (бинарный)	Вычитание
5	=,*=,%=,+=,-= (специальные)	Операторы присвоений

Как видите, операторы умножения, деления, деления по модулю, сложения и вычитания имеют вид обычных математических операций. Да они и работают так же, как соответствующие им арифметические операции:

```
float var = 133 / 12;
```

Значение большинства операторов вам хорошо известно еще из начальной школы, кроме разве что операции деления по модулю.

По своей сути этот оператор означает получение остатка от деления. Например, 4 входит в 15 три раза, и остаток при этом составляет 3. Выражаясь терминами C++, 15, деленное по модулю 4, равно 3.

```
int var = 15 % 4; //переменной var присваивается значение 3
```

Программисты всегда пытаются удивить непрограммистов, а потому в C++ деление по модулю определяется так:

```
IntValue % IntDivisor
```

эквивалентно

```
IntValue - (IntValue / IntDivisor)* IntDivisor
```

Вот пример:

```
15 % 4 равно    15 - (15/4) * 4
                15 - 3*4
                15 - 12
                3
```



Для действительных переменных оператор деления по модулю не определен, поскольку он целиком основан на использовании округления (округления рассматривались в главе 2, "Премудрости объявления переменных").

Анализ выражений

Самым распространенным видом инструкций в C++ является выражение. *Выражением* в C++ называют любую последовательность операторов (длиной не меньше одного), которая возвращает значение. Все выражения типизированы. Тип выражения определяется типом возвращаемого значения. Например, значение выражения 1 + 2 равняется 3, следовательно, это целочисленное выражение (тут нужно вспомнить, что константы без дробной части определяются как имеющие тип int). Синтаксическая конструкция, включающая математический оператор, является выражением, так как в результате выполнения любой операции получается число.

Выражения бывают как сложными, так и крайне простыми. Необычной особенностью C++ является то, что он понимает под выражением любой законченный оператор. Поэтому корректным оператором является, например, `1;`.

Он тоже представляет собой выражение, потому что его значение 1, а тип `int`. В операторе

```
z = x * y + w;
```

можно выделить пять выражений:

```
x * y + w
x * y
x
y
w
```

Необычный аспект C++ состоит в том, что выражение само по себе является завершенной инструкцией, т.е. упомянутое выражение `1;` — завершенная инструкция C++.

Определение порядка операций

Все операторы выполняют определенные функции. Чтобы установить порядок выполнения различных операторов, им назначены приоритеты. Рассмотрим выражение

```
int var = 2 * 3 + 1 ;
```

Если сложение выполнить перед умножением, то значением выражения будет 2 * 4 = 8. Если сперва выполнить умножение, то получим значение 6 + 1 = 7.

Приоритеты операций определяют порядок выполнения вычислений. Из табл. 3.1 видно, что приоритет операции умножения выше, чем сложения, т.е. результат все же равен 7 {приоритеты используются и в арифметике, и C++ следует именно им).

А что происходит, когда в одном выражении используется два оператора с одинаковым приоритетом?

```
int var = 8 / 4 / 2;
```

Как в этом случае следует поступить: сначала 8 поделить на 2 или 2 на 4? Если в одном выражении присутствуют операции с одинаковыми приоритетами, они выполняются слева направо (то же правило применяется и в арифметике). Поэтому в предыдущем примере сперва делим 8 на 4, получая 2, а затем делим его на 2, получая ответ — 1.

В выражении

```
x / 100 + 32
```

x делится на 100 и к результату добавляется 32. Но что, если программисту нужно поделить x на сумму 100 и 32? В таком случае ему придется использовать скобки:

```
x / (100 + 32)
```

При вычислении такого выражения x будет делиться на 132.

Заметим, что начальное выражение

```
x / 100 + 32
```

идентично следующему:

```
(x / 100) + 32
```

Почему это действительно так? Потому что C++ сначала выполняет операции с высшим приоритетом. А приоритет операций умножения и деления выше, чем сложения и вычитания. Поэтому скобки, указывающие на высокий приоритет данной операции, можно опустить.

На основе сказанного можно сделать вывод: в случае необходимости приоритет оператора можно повысить, используя скобки.

Выполнение унарных операций

С арифметическими бинарными операторами вы неоднократно встречались с самого первого класса. О существовании же унарных операций вы могли и не подозревать, хотя наверняка одну из них использовали довольно часто (имея дело с отрицательными числами).

Унарными называются те операторы, которые имеют только один аргумент, например *-a*.

Унарными математическими операторами являются +, -, ++ и --. Рассмотрим некоторые из них:

```
int var1 = 10;
int var2 = -var1;
```

Здесь в последнем выражении используется унарный оператор "-".

Оператор "минус" изменяет знак своего аргумента (в примере это var1) на противоположный. Положительные числа становятся отрицательными и наоборот. Оператор "плюс" знака аргумента не изменяет и фактически вообще ни на что не влияет.

Операторы ++ и -- вы можете увидеть впервые. Они увеличивают или уменьшают на 1 значение аргумента и поэтому называются операторами инкремента и декремента (от англ. *increment* (увеличивать) и *decrement* (уменьшать). — *Прим. перев.)*. К действительным переменным их применение недопустимо. После выполнения приведенного ниже фрагмента значение переменной будет равно 11.

```
int var = 10; // Инициализация переменной
    var++;    // Ее увеличение; значение переменной равно 11
```

Операторы инкремента и декремента могут находиться либо перед аргументом (префиксная форма), либо после него (постфиксная форма). В зависимости от способа записи, выполнение операторов инкремента и декремента имеет свои особенности. Рассмотрим оператор инкремента (принципы работы оператора декремента те же).

Предположим, что переменная п имеет значение 5. Оба способа применения к п оператора инкремента (++п и п++) приведут к результату 6. Разница между ними состоит в том, что значение п в выражении ++п равно 6, в то время как в выражении с постфиксной формой записи оно равно 5. Это можно проиллюстрировать следующим примером:

```
             //объявляем три целые переменные
int n1, n2, п3;

n1 = 5;
n2 = ++n1; //обе переменные - n1 и п2 - получают значение 6

nl = 5;
п3 = nl++; // nl принимает значение 6, а п3 - 5
```

Другими словами, переменной г.2 присваивается уже увеличенное префиксным оператором инкремента значение nl, тогда как переменной n3 передается еще не увеличенное постфиксным оператором значение n1.

Использование операторов присвоения

Операторы присвоения являются бинарными, изменяющими значения своих левых аргументов. Обыкновенный оператор присвоения "=" абсолютно необходим во всех языках программирования. Этот оператор сохраняет значение правого аргумента

в левом. Однако причуды авторов языка привели к появлению и других операторов присвоения.

Создатели C++ заметили, что присвоение часто имеет вид

```
variable = variable # constant
```

Здесь # представляет собой какой-то бинарный оператор. Следовательно, чтобы увеличить целую переменную на два, программист может написать:

```
nVariable = nVariable + 2;
```

Из этой записи следует, что к значению переменной nVariable добавляется двойка и результат снова сохраняется в nVariable.



Использование в левой и правой части выражения одной и той же переменной весьма распространенное явление в программировании.

Поскольку одна и та же переменная находится по обе стороны знака равенства, было решено просто добавить оператор, используемый при вычислении, к знаку присвоения. В таких специфических операторах присвоения допускается использование всех бинарных операторов. Поэтому указанное выше выражение можно сократить до

```
nVariable += 2;
```

Смысл этой записи таков: "значение переменной nVariable увеличено на 2".

Почему так важен оператор инкремента

Разработчики C++ заметили, что программисты прибавляют 1 чаще, чем любую другую константу. Учитывая это, в язык была добавлена соответствующая конструкция.

Кроме того, большинство процессоров способны выполнять команды инкремента быстрее, чем команды сложения. Учитывая мощность микропроцессоров, которые использовались во время создания C++, подобное нововведение было действительно важным.



Модифицированные операторы присвоения используются не так часто, как обычные, но как правило повышают удобочитаемость программ.

Глава 4

Выполнение логических операций

В этой главе...

- ✓ Зачем нужны логические операторы
- ✓ Использование простых логических операторов
- ✓ Бинарные числа в С++
- ✓ Выполнение битовых логических операций

Наиболее распространенной синтаксической конструкцией С++ является выражение. Большинство используемых выражений содержит арифметические операторы сложения (+), вычитания (-) и умножения {*). В данной главе описаны все эти типы выражений.

Существует целый класс так называемых *логических* операторов. В отличие от арифметических, этот тип операторов многими не воспринимается как операторы.

Неправда, что люди не сталкиваются с логическими операторами. Значения операторов И и ИЛИ они вычисляют постоянно. Я не буду есть овсянки без молока И сахара. И закажу себе ром ИЛИ шотландский виски. Как видите, люди очень часто используют логические операторы, не осознавая этого.

Логические операторы бывают двух типов. Операторы И и ИЛИ называются простыми логическими операторами. Операторы второго типа, или битовые операторы, уникальны, так как используются только в программировании. Этот тип операторов позволяет работать с любым битом в машинном представлении числа.

Зачем нужны логические операторы

У вас может возникнуть вопрос: "Если до сегодняшнего дня меня совершенно не волновали логические операторы, почему это должно случиться теперь?". Да потому, что программы должны "уметь" принимать решения. Программы, написанные без принятия решений, по сложности подобны приведенной в первой главе (вспомните, что все выполняемые ею действия совершенно безальтернативны). Для принятия решений в программах просто необходимо использовать логические операторы.

Использование простых логических операторов

Программы на С++ должны обладать способностью принимать решения. Программа Convert не выполняла ничего, кроме простого преобразования значений температуры, и не принимала никаких решений, основанных на входных значениях. Для принятия таких решений в программах С++ используют логические операторы.

Простые логические операторы приведены в табл. 4.1. Они могут возвращать два значения: `true` (истина) и `false` (ложь).

Таблица 4.1. Простые операторы из повседневной логики

ОПЕРАТОР	ЗНАЧЕНИЕ
==	Равенство; истинно, когда значение левого аргумента совпадает со значением правого
!=	Неравенство; противоположно равенству
> , <	Больше, меньше; истинно, когда значение левого выражения больше (или меньше) значения правого
>= , <=	Больше или равно, меньше или равно; истинно, если истиной является > или == (соответственно < или ==)
&&	И; истинно, если аргументы и слева и справа являются истиной
||	ИЛИ; истинно, если или левый, или правый аргумент являются истиной
!	НЕ; истинно, если его аргумент принимает ложное значение

Первые шесть операторов табл. 4.1 являются операторами сравнения. Оператор равенства используется для проверки равенства двух значений. Например, следующее выражение истинно, если значением п является 0, и ложно во всех других случаях:

```
п == 0;
```



Не перепутайте оператор равенства == с оператором присвоения =. Эта ошибка очень распространена, к тому же компилятор C++, вообще говоря, не считает ее ошибкой, что делает ее вдвойне опасной!

```
п = 0; // Программист хотел написать, что п == О
```

Широко распространены в повседневной жизни операторы "больше" (>) и "меньше" (<). Приведем пример логического выражения, возвращающего значение true.

```
int nl = 1;
int n2 = 2;
nl < n2
```

Операторы "больше" и "меньше" внешне очень похожи, и поэтому их легко перепутать. Чтобы этого не случилось, помните, что оператор-стрелочка принимает значение true в том случае, когда из двух сравниваемых значений он указывает на меньшее.

С помощью операторов > и < можно найти случаи, когда nl больше или меньше п2, однако при этом игнорируется возможность равенства их значений. Операторы "больше или равно" (>=), "меньше или равно" (<=), в отличие от только что рассмотренных, учитывают и эту возможность.

Так же широко используются и операторы && (И) и || (ИЛИ). Эти операторы обычно сочетаются с другими логическими операторами:

```
// истинно, если п2 больше nl и меньше п3
(nl < n2) && (п2 < п3);
```

В качестве ремарки: оператор "больше или равно" можно определить как[10]

```
nl <= n2 эквивалентно (nl < n2) || (nl == n2)
```

Логические операции и действительные переменные

Переменные с плавающей точкой, как уже отмечалось, не могут использоваться для перечисления. Вы можете сказать: первый, второй, третий, четвертый и т.д., так

[10] *В качестве еще одной ремарки: операторы сравнения вообще достаточно взаимозаменяемы. Так, например,* (a==b) *эквивалентно* (!(a>b) &&!(a<b)).— Прим. ред.

как соотношения между 1, 2, 3 абсолютно точно известны. Но нет никакого смысла говорить о номере 4.535887 в последовательности (такой способ нумерации возможен лишь как обозначение чего-то между четвертым и пятым, но не действительного значения номера, так как в любом сколь угодно малом отрезке их несчетное[11] множество).

Тип float, представляющий в C++ действительные числа, не является перечислимым. Кроме того, в отличие от действительных чисел, числа с плавающей точкой имеют ограниченное количество разрядов, поэтому при использовании операторов сравнения с числами с плавающей точкой необходимо соблюдать осторожность. Рассмотрим следующий пример:

```
float f1 = 10.0;
float f2 = f1 / 3;
f1 == (f2 * 3.0); // Равны ли эти значения?
```

Сравнивая начальное и полученное значения, мы не обязательно получим равенство. Действительные переменные, с которыми работает компьютер, не могут содержать бесконечного числа значимых разрядов. Поэтому f2 равняется, например, 3.3333, а не $3\frac{1}{3}$. В отличие от математики, в компьютере число троек после точки ограничено. Умножив 3.3333 на 3, вы, вероятно, получите не 10.0, а 9.9999. Такой маленькой разницей может пренебречь человек, но не компьютер. Эта машина понимает под равенством исключительно точное равенство значений.

В современных процессорах выполнение таких вычислений очень усложнено. Процессор может даже устранить ошибку округления, но точно определить, когда именно процессору вздумается это сделать, язык C++ не способен.

Проблемы могут появиться и при совершенно простых вычислениях, например:

```
float f1 = 10.0;
float f2 = 120 % 11;
f1 == f2;    //истинно ли это выражение?
```

Теоретически f1 и f2 должны быть равны (об операции деления по модулю можно прочитать в главе 3, "Выполнение математических операций"). Ошибка округления возникнуть вроде бы не должна. Однако и в этом нельзя быть уверенным: вам ведь неизвестно, как именно представляются числа с плавающей точкой внутри компьютера.

Позвольте порекомендовать более безопасное сравнение:

```
float f1 = 10.0;
float f2 = f1 / 3;
float f3 = f2 * 3.0;
(f1 - f3) < 0.0001 SS (f3 - f1) < 0.0001;
```

Оно ИСТИННО В ТОМ случае, если разница между f1 и f2 меньше какого-то малого значения (в нашем случае — 0.0001); при этом небольшие погрешности вычислений на правильность сравнения не повлияют.

Сокращенные вычисления в C++

Рассмотрим следующую конструкцию:

```
условие1 && условие2
```

Если условие1 ложно, то результат не будет истинным, независимо от истинности выражения условие2. В схеме

```
условие1 || условие2
```

в случае истинности выражения условие1 неважно, какое значение принимает условие2, — результат будет истинным.

---

[11] *Более того, в данном случае это не красивое слово, а строгий математический термин.* — Прим. ред.

Для экономии времени С++ вычисляет первым условие]., и в случае, если оно ложно (для оператора `&&`) или истинно (для оператора `||`), выражение `условие2` не вычисляется и не анализируется.

Типы логических переменных

Поскольку > является оператором, то сравнение а > 10 представляет собой выражение. Очевидно, что результатом такого выражения может быть или true (истина), или false (ложь).

Вы уже могли заметить, что среди обсуждаемых в главе 2, "Премудрости объявления переменных", типов переменных не было логических (булевых) типов. Более того, ни в одном из существующих типов С++ нет значений true или false. Тогда к какому типу отнести выражение а > 10?

В С++ для хранения логических значений используется тип int. При этом 0 обозначает false, а любое другое отличное от нуля значение является истиной (true). Выражения типа а > 10 могут принимать значения 0 (false) или, например, 1 (true).



Microsoft Visual Basic для интерпретации значений false и true также использует целые числа, но в нем операторы сравнения возвращают или 0 (false), или -1 (true).

В новом стандарте ANSI С++ для работы с булевыми переменными определен тип `bool`, имеющий значения true и false.

'Бшифнме чисяа С++

Переменные хранятся в компьютере в виде так называемых двоичных, или бинарных, чисел, т.е. представлены в виде последовательности битов, каждый из которых может содержать два значения: 0 или 1. Скорее всего, вам не придется оперировать с числами на битовом уровне, хотя существуют ситуации, когда обойтись без этого нельзя. С++ снабжен несколькими операторами для подобных целей.



Вряд ли вам придется часто работать с переменными на битовом уровне, поэтому остальную часть главы следует рассматривать как техническое отступление от основного повествования.

Так называемые битовые логические операторы работают с аргументами на битовом уровне. Для того чтобы понять принципы их работы, давайте рассмотрим, как компьютер хранит переменные.

Десятичная система счисления

Числа, которыми мы чаще всего пользуемся, называются *десятичными*, или числами по основанию 10. В основном программисты на С++ тоже используют десятичные переменные. Например, мы говорим, что значение переменной var равно 123.

Число 123 можно представить в виде `1*100+2*10+3*1`. При этом каждое из чисел 100, 10, 1 является степенью 10.

$$123 = 1 * 100 + 2 * 10 + 3 * 1,$$

что эквивалентно следующему:

$$123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

Помните, что любое число в нулевой степени равняется 1.

Другие системы счисления

Использование числа 10 в качестве основания нашей системы счисления объясняется, по всей вероятности, тем, что издревле для подсчетов человек использовал пальцы рук. Учитывая особенности нашей физиологии, удобной альтернативной системой счисления можно было бы выбрать двадцатеричную[12] (т.е. с основанием 20).

Если бы наша вычислительная система была заимствована у собак, то она бы была восьмеричной (еше один "разряд", находящийся на задней части каждой лапы, не учитывается). Эта система счисления работала бы не менее хорошо:

$123_{10} = 1 * 8^2 + 7 * 8^1 + 3 * 8^0 = 173_8$

Индексы 10 и 8 указывают систему счисления: 10 — десятичная. 8 — восьмеричная. Основанием системы счисления может быть любое положительное число.

Двоичная система счисления

У компьютеров, видимо, пальцев поменьше (может быть, поэтому они такие недалекие?). Они предпочитают пользоваться двоичной системой счисления. Число 123 переводится в двоичную систему таким образом:

$123_{10} = 0*128 + 1*64 + 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1 = 01111011_2$

Существует соглашение, которое гласит, что в записи двоичных чисел используются 4, 8, 16 или 32 (и т.д.) двоичных цифр, даже если старшие цифры — нули. Внутреннее представление числа в компьютере строится именно таким образом.

Понятие разряда применяется к числам, кратным десяти, двоичный же разряд называется битом. Восемь битов составляют байт, а слово представляется или двумя, или четырьмя байтами.

Поскольку основа двоичной системы счисления очень мала, для представления чисел необходимо использовать слишком большое количество битов. Для представления таких обычных чисел, как 123_{10}, неудобно использовать выражения вида 01111011_2. Поэтому программисты предпочитают представлять числа блоками из четырех битов.

С помощью одного четырехбитового блока можно представить любое число от 0 до 15, и такая система счисления называется шестнадцатеричной (*hexadecimal*), т.е. системой по основанию 16. Часто употребляют ее сокращенное название *hex*.

В шестнадцатеричной системе обозначения цифр от 0 до 9 остаются теми же, а числа от 10 до 15 представляются с помощью первых шести букв алфавита: А вместо 10, в вместо 11 и т. д. Следовательно, 123_{10} — это $7B_{16}$.

$123 = 7 * 16^1 + B(\text{т.е. } 11) * 16^0 = 7B_{16}$

Поскольку программисты предпочитают представлять числа с помощью 4, 8, 16 или 32 битов, шестнадцатеричные числа состоят соответственно из 1, 2, 4 или 8 шестнадцатеричных разрядов (даже если ведущие разряды равны 0).

В заключение замечу, что, так как терминал не поддерживает нижний индекс, записывать шестнадцатеричные символы в виде $7B_{16}$ неудобно. Даже в том текстовом редакторе, который я использую сейчас, довольно неудобно всякий раз менять режимы шрифтов для ввода всего двух символов. Поэтому программисты договорились начинать шестнадцатеричные числа с 0x (это странное обозначение было придумано еще во время разработки языка С). Таким образом, $7B_{16}$ равно 0x7в. Следуя этому соглашению, 0x7B равно 123, тогда как 0x123 равно 291.

К шестнадцатеричным числам можно применять все те же математические операторы, что и к десятичным. Нам трудно выполнить в уме умножение чисел Cxc*CxE потому, что таблица умножения, которую мы учили в школе, применима только к десятичной системе счисления.

---

[12] *Что и было сделано у некоторых народов, например у майя или чукчей.* — Прим. ред.

Выражения с римскими числами

Интересно, что некоторые системы чисел значительно препятствовали развитию математики, К таким относится и так называемая римская система.

Сложить два римских числа не очень сложно:

XIX + XXVI = XXV

Последовательность выполнения сложения такова:

а) IX+VI: I после V "уничтожает" I перед x, поэтому в результате получаем XV;

б) X+XX=XXX, если добавить еще один x, получим xxxx, или XL.

Сложность вычитания римских чисел приблизительно такая же. Однако, чтобы умножить два римских числа, требуется диплом бакалавра математики (у вас волосы станут дыбом от правила, которое объясняет, как добавить к X разряд справа так, чтобы X*IV равнялось XL). А уж о делении римских чисел можно писать целые докторские диссертации...

Выполнение битовых логических операций

Все числа C++ могут быть представлены в двоичном виде, т.е. с использованием только 0 и I в записи числа. В табл. 4.2 указаны операции, которые работают с числами побитово; отсюда и происходит название термина "битовые операции".

Таблица 4.2. Битовые операции

ОПЕРАТОР	ФУНКЦИЯ
~	Каждый бит меняет свое значение на противоположное: 0 заменяется 1, 1 — нулем
&	Битовое И: поочередно выполняет операцию И с парами битов левого и правого аргумента
|	Битовое ИЛИ
^	Битовое исключающее ИЛИ

С помощью битовых операций можно сохранять большое количество информации в маленьком участке памяти. В мире существует множество вещей, которые имеют только два состояния (или, максимум, четыре). Вы или женаты (замужем), или нет (хотя можете быть в разводе или еще не женаты). Вы или мужчина, или женщина (по крайней мере, так сказано в моих водительских правах). В C++ каждую такую характеристику вы можете сохранить в одном бите. Таким образом, поскольку для хранения целого числа выделяется 4 байта, в тип int можно упаковать значения 32 разных свойств.

Кроме того, битовые операции выполняются крайне быстро. Хранение 32 характеристик в одном типе не приводит ни к каким дополнительным затратам.

Битовые операции с одним битом

Битовые операторы &, | и ~ выполняют логические операции над отдельными битами числа. Если рассматривать 0 как false и 1 как true (так принято, хотя можно ввести и другие обозначения), то для оператора ~ справедливо следующее:

~1(true) равно 0(false)
~0 (true) равно 1 (false)

Оператор & определяется так:

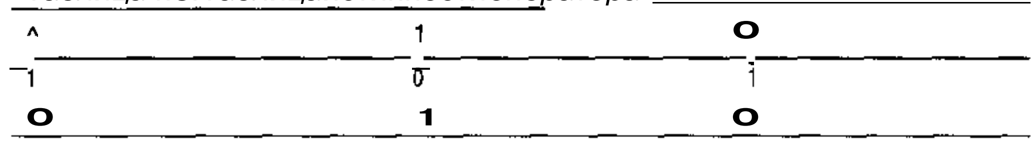
```
1(true)  & 1 (true)  равно 1(true)
1 (true)  & 0 (false) равно 0(false)
0(false) 5 0 (false) равно 0(false)
0 (false) & 1 (true)  равно 0(false)
```

Для оператора |:

```
1(true)   | 1 (true)   равно 1(true)
1 (true)  | 0 (false)  равно 1(true)
0 (false) | 0 (false)  равно 0(false)
0 (false) ! 1 (true)   равно 1(true)
```

Для последнего логического оператора, называемого "исключающим или" (XOR), прямой аналог в повседневной жизни найти труднее. Он возвращает значение true, если истинным является какой-то один (но не оба!) из его аргументов. Таблица истинности этого оператора представлена ниже (табл. 4.3).

Таблица 4.3. Таблица истинности оператора ^



^	1	0
1	0	1
0	1	0

Теперь, зная, как работают эти операторы с отдельными битами, рассмотрим их применение к двоичным числам.

Использование битовых операторов

Битовые операторы выполняются подобно любым другим арифметическим операторам. Самым легким для понимания является оператор ~. Выполнить операцию ~ над числом означает выполнить ее над каждым битом числа.

~0110_2 (0x6)
1001_2 (Cx9)

Таким образом получаем, что ~0x6 равно 0x9.

В следующем примере продемонстрировано выполнение оператора &:

0110_2
&
OOHг
0010_2

Вычисляем, начиная со старших битов: 0 & 0 равно 0. В следующем бите 1 & 0 равно 0. В третьем бите 1 & 1 дает 1, а в последнем бите 0 & 1 дает 0.

Те же вычисления могут быть выполнены в шестнадцатеричной системе. Для этого нужно преобразовать числа в двоичное представление, выполнить операцию и преобразовать результат обратно.

0x06	0110_2
&	&
0x03	0011_2
0x02	0010_2

Расписав числа таким образом, мы получили, что Oxб & 0x3 равно 0x2.

(Попробуйте подсчитать значение выражения 0x6 | 0x3. Если вам это удастся, вы почувствуете себя на седьмом небе. Иначе очутитесь на первой из семи ступенек в преисподнюю. У меня на это ушло чуть меньше восьми минут.)

Простой пример

Следующая программа иллюстрирует работу побитовых операторов. В ней инициируются две переменные, к которым применяются операции &, |, ^. Результаты вычислений выводятся на экран.

```
// BitTest — инициируются две переменные и
//           выводятся результаты выполнения
//           операторов ~,&,| и ^
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // установка вывода в шестнадцатеричном виде
    cout.setf(ios::hex, ios::hex);
    // инициализация двух аргументов
    int nArg1 = 0x1234;
    int nArg2 = 0x00ff;

    // Выполнение логических операций
    // Сначала применяем унарный оператор NOT
    cout << "Arg1          = 0x" << nArg1 << '\n";
    cout << "Arg2          = 0x" << nArg2 << "\n";
    cout << "~nArg1        = 0x" << ~nArg1 << "\n";
    cout << "~nArg2        = 0x" << ~nArg2 << "\n";

    // Теперь - бинарные операторы
    cout << "nArg1 & nArg2 = 0x"
         << (nArg1 s nArg2)
         << "\n";
    cout << "nArg1 | nArg2 = 0x"
         << (nArg1 | nArg2)
         << "\n";
    cout << "nArg1 ^ nArg2 = 0x"
         << (nArg1 ^ nArg2)
         << "\n";

    returnC);
}
```

Первая инструкция нашей программы (та, которая следует за ключевым словом main) — `cout.setf(ios::hex);` — меняет используемый по умолчанию десятичный формат вывода на шестнадцатеричный (поверьте мне, это сработает).

В остальном программа очевидна. Присвоив значения аргументам nArgl, nArg2, она выводит все варианты побитовых вычислений над этими переменными.

Результат выполнения программы будет выглядеть следующим образом:

```
Arg1          = 0x1234
Arg2          = 0xff
~nArg1        = 0xffffedcb
~nArg2        = 0xffffff00
nArg1 S nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
```



К шестнадцатеричным числам всегда приписывается префикс 0x.

Зачем придуманы эти глупые операторы

Предназначение большинства операторов очевидно. В необходимости операторов - или + сомневаться не приходится. Не нужно рассказывать, для чего используется оператор ||. Но для начинающих программистов может быть далеко не очевидно, зачем использовать битовые операторы.

Оператор & часто используется для маскирования информации. Например, предположим, что нам нужно выделить последний значимый шестнадцатеричный разряд из четырехразрядного числа:

```
0x1234          0001 0010 OOH 0100
&               &
OxOOOF          0000 0000 0000 1111
0x0004          0000 0000 0000 0100
```

С помощью этого оператора можно также выделять и устанавливать значения отдельных битов.

Представьте себе, что в написанной вами базе данных для сохранения некоторой информации о личности используется единственный бит. Первый значимый бит равен 1. если это особа мужского пола, второй бит равен 1, если это программист, третий равен 1 в случае внешней привлекательности, а четвертый — если человек имеет собаку. Взгляните на табл. 4.4.

Таблица 4.4. Значения битов

Бит	ЗНАЧЕНИЕ
0	1 → мужчина
1	1 → программист
2	1 → привлекательный
3	1 → владелец собаки

Не особо привлекательный, имеющий собаку программист мужского рода будет закодирован числом 1101_2. Если вы хотите просмотреть все записи в поисках хорошенькой девушки без собаки, причем неважно, разбирается ли она в С++, надо использовать следующее сравнение (числа приведены в двоичной записи!):

```
(databaseValue & 1011) == 0010
                             ^    - не мужчина
                               ^  - привлекательна
                                ^ - без собаки
                       ^ ^^- важно
                        *   - не интересует
```

В этом случае число 1011 является маской, потому что оно маскирует, исключая из рассмотрения, биты с не интересующими вас характеристиками.

Глава 5

Операторы управления программой

В этой главе...

- ✓ Управление ходом программы с помощью команд ветвления
- ✓ Выполнение циклов
- ✓ Вложенные команды управления
- ✓ Инструкция выбора

Простые программки, которые появлялись в первых четырех главах, обрабатывали фиксированное количество вводов и выводов результатов вычислений, после чего завершали работу. Эти приложения были лишены какого бы то ни было контроля над работой программы, в частности не выполняли никаких проверок Но компьютерные программы могут принимать решения. Вспомните: когда пользователь нажимает клавиши, компьютер реагирует на это выполнением соответствующих команд.

Например, если пользователь нажимает <Ctrl+C>, компьютер копирует содержимое выделенного блока в буфер обмена. Если пользователь перемещает мышь, на экране двигается ее курсор. Этот список можно продолжать до бесконечности. Программы же, которые не принимают никаких решений, неизбежно скучны и однообразны.

Команды, управляющие ходом программы, указывают на то, какие действия она должна выполнить в зависимости от результата вычисления какого-либо логического выражения (о которых шла речь в главе 4, "Выполнение логических операций"). Существует три типа управляющих инструкций: операторы ветвления (или условного перехода), цикла и выбора.

Управление ходом программы с помощью команд ветвления

Проще всего управлять ходом программы с помощью инструкции ветвления, которая позволяет программе, в зависимости от результата логического выражения, решить, по какому из двух возможных путей выполнения инструкций следует двигаться дальше.

В C++ оператор условного перехода реализуется с помощью инструкции `if`:

```
if (m > n)
{
//1-я последовательность операторов
//инструкции, которые должны быть выполнены,
//если m больше n
}
else
{
```

```
//2-я последовательность операторов
//инструкции, которые нужно выполнить
//в противном случае
}
```

Прежде всего вычисляется логическое выражение m > п. Если его значение — true, программа выполняет первую последовательность операторов. Если же выражение ложно, управление передается второй последовательности. Оператор else не обязателен: если он опушен, С++ считает, что он существует, но является пустым.



Если в текущей ветви оператора if имеется только одна инструкция, скобки использовать необязательно. Однако очень легко сделать ошибку, которую без скобок, определяющих структуру операторов, компилятор С++ обнаружить не сможет. Поэтому намного безопаснее включать скобки всегда. Если друзья будут уговаривать вас не использовать скобки, не поддавайтесь!

Работу оператора if можно рассмотреть на следующем примере:

```
// BranchDemo — введите два числа. Если первый
// аргумент больше, выполняем операторы первой
// ветви, если меньше — второй

#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
(
    // вводим первый аргумент...
    int arg1;
    cout << "Введите arg1: ";
    cin  >> arg1;

    // ...второй
    int arg2;
    cout << "Введите arg2: ";
    cin  >> arg2;

    // теперь решаем, что делать:
    if (arg1 > arg2)
    {
        cout << "аргумент 1 больше, чем аргумент 2\n";
    }
    else
    {
        cout << " аргумент I не больше, чем аргумент 2\п";
    }

    return 0;
}
```

Программа считывает два целых числа, вводимых с клавиатуры, и сравнивает их. Если выражение "argl больше arg2" истинно, то выполняется инструкция cout « "аргумент 1 больше, чем аргумент 2\n";. Если же нет, то управление переходит к последовательности операторов, соответствующей условию else: cout << "аргумент 1 не больше, чем аргумент 2\г.";.

Выполнение циклов

Оператор перехода позволяет управлять работой программы, когда существуют альтернативные пути ее выполнения. Это усовершенствование языка хотя и весьма значительное, но все же не достаточное для написания полнофункциональных программ.

Рассмотрим проблему обновления экрана компьютера. При перерисовывании содержимого типичного дисплея компьютеру необходимо выводить на экран тысячи пикселей. Если программа не умеет повторно выполнять один и тот же фрагмент кода, вы будете вынуждены тысячи раз записывать одно и то же множество инструкций.

Для решения этой проблемы необходим способ, который позволит многократно выполнять одни и те же последовательности инструкций. Операторы цикла предоставляют возможность решить эту задачу.

Цикл while

Самый простой цикл можно организовать с помощью оператора while. Он выглядит таким образом:

```
while (условие)
{
//Этот код выполняется повторно,
//пока условие остается истинно
}
```

Сначала проверяется условие. Условием могут быть выражения вида var > 10, var1 == var2 или любые другие. Если условие истинно, выполняются инструкции в скобках. Дойдя до закрывающей скобки, компилятор передает управление в начало цикла, и все повторяется вновь. Таким образом, смысл оператора while в том, что программный код в скобках повторно выполняется до тех пор, пока не нарушится условие (этот процесс напоминает мне утренние прогулки с собакой вокруг дома, пока она не... ну а потом мы возвращаемся).

Если условие сначала было справедливо, тогда что может заставить его стать ложным? Рассмотрим следующий пример программы:

```
// WhileDemo — введите счетчик цикла.
// Программа выводит количество выполненных
// циклов while
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{
    // ввод счетчика цикла
    int loopCount;
    cout << "Введите loopCount: ";
    cin >> loopCount;

    // теперь в цикле выводим значения
    while (loopCount > 0)
    {
        loopCount = loopCount - 1;
        cout << "Осталось выполнить "
             << loopCount << " циклов\n";
```

```
    }
    return 0;
}
```

Программа WhileDemo получает от пользователя значение счетчика цикла, которое сохраняется в переменной loopCount. Затем программа выполняет цикл while. Сначала проверяется значение переменной loopCount. Если оно больше нуля, программа входит в тело цикла (телом цикла называется код между скобками), где loopCount уменьшается на 1, и результат выводится на экран. Затем программа возвращается к началу цикла и проверяет, осталась ли переменная loopCount положительной.

Ниже представлены результаты выполнения программы, выведенные на экран. Нетрудно догадаться, что введенный мною счетчик цикла равен 5. Программа пять раз выполнила цикл, каждый раз выводя результат на экран:

```
Осталось выполнить 4 циклов
Осталось выполнить 3 циклов
Осталось выполнить 2 циклов
Осталось выполнить 1 циклов
Осталось выполнить 0 циклов
```

Если пользователь введет отрицательное значение для счетчика цикла, условие окажется ложным и тело цикла не будет выполнено ни разу. Если пользователь введет очень большое число, на выполнение программы уйдет очень много времени.

Реже используется другая версия цикла while, известная как do. . .while. Она работает аналогично, но условие завершения проверяется в конце, после выполнения тела цикла.

```
do
{
    //Тело цикла
} while (условие);
```

Поскольку условие проверяется только в конце, тело оператора *do. . .*while выполняется всегда хотя бы один раз.



Условие завершения цикла проверяется только в начале оператора while или в конце оператора do. . .while. Даже если в какой-то момент оно перестанет быть справедливым, программа продолжит выполнение цикла до следующей проверки условия.

Использование операторов инкремента и декремента

Очень часто для какого-либо подсчета в циклах программисты используют операторы инкремента или декремента. Заметим, что в следующем фрагменте программы WhileDemo для уменьшения значения счетчика используются операторы присвоения и вычитания:

```
// в цикле вызолим значения
    while (loopCount > 0)
    {
        loopCount = loopCount — 1;
        cout « "Осталось выполнить "
             << loopCount << " циклов\n";
    }
```

Используя оператор декремента, этот цикл можно записать более компактно:

```
// в цикле выводим значения
    while (loopCount > 0)
    {
        loopCount--;
        cout << "Осталось выполнить "
             « loopCount << " циклов\п";
    }
```

Смысл этого варианта цикла полностью совпадает со смыслом оригинала. Единственная разница — в способе записи.

Поскольку оператор декремента одновременно уменьшает аргумент и возвращает его значение, он может включаться в условие цикла while. В частности, допустима следующая версия цикла:

```
// в цикле выводим значения
    while (loopCount-- > 0)
    (
        cout << "Осталось выполнить "
             << loopCount << " циклов\п";
    }
```

Хотите — верьте, хотите — нет, но большинство программистов на C++ используют именно этот вариант записи. И не потому, что им нравится быть остроумными; хотя почему бы и нет? Использование в логических сравнениях операторов инкремента и декремента делает программный код легко читаемым и более компактным. И вряд ли вы, исходя из своего опыта, сможете предложить достойную альтернативу.



И в выражении loopCount--, и в --loopCount значение loopCount уменьшается; однако первое возвращает значение переменной loopCount перед его уменьшением на 1, а второе — после.

Сколько раз будет выполняться декрементированный вариант WhileDemo, если пользователь введет число 1? Если использовать префиксный вариант, то значение --loopCount равно 0 и тело цикла никогда не выполнится. В постфиксном варианте loopCount-- возвратит 1 и программа передаст управление в начало цикла.

У вас может сложиться ошибочное мнение, что программа, в которой используются операторы декремента, выполняется быстрее, так как содержит меньше инструкций. Однако это не совсем так. Время выполнения программы не зависит от того, какую из представленных выше операций декремента вы используете, так как современные оптимизирующие компиляторы используют минимально необходимое количество инструкций машинного языка, независимо от используемых для декремента операторов.

Использование цикла for

Другой разновидностью циклов является цикл for. Его часто предпочитают более простому циклу while. Цикл for имеет следующий вид:

```
for (инициализация; условие; увеличение)
i
    //...тело цикла
}
```

Выполнение цикла for начинается с инициализации. В ней обычно находится оператор присвоения, используемый для установки начального значения перемен-

ной цикла. Условие инициализации выполняется только один раз, при первом входе в цикл for.

Затем проверяется условие. Подобно циклу while, цикл for выполняется до тех пор, пока условие не станет ложным.

После того как выполнится код тела цикла, управление получит следующий параметр цикла for (увеличение) и значение счетчика изменится. Затем опять будет выполнена проверка условия, и процесс повторится. В этом параметре обычно записывают инкрементное или декрементное выражение, которое определяет характер изменения переменной цикла на каждой итерации, но в принципе ограничений на используемые здесь операторы нет.

Цикл for можно заменить эквивалентным ему циклом while:

```
инициализация;
while (условие)
{
    {
        // ... тело цикла
    }
    увеличение;
}
```

Все три параметра цикла for являются необязательными. C++ игнорирует отсутствие части инициализации или увеличения цикла, а если опущено условие, C++ будет выполнять цикл for вечно (или пока какой-либо другой оператор не передаст управление за пределы цикла).

Для лучшего понимания цикла for рассмотрим пример. Приведенная ниже программа ForDemo выполняет то же, что и whileDemo, но вместо while использует цикл for.

```
// ForDemo. Вводится счетчик цикла.
// На экран выводится количество выполненных
// циклов for
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{

    // ввод счетчика цикла
    int loopCount;
    cout « "Введите loopCount: ";
    cin >> loopCount;

    // работаем, пока не нарушится условие
    for (int i = loopCount; i > 0; i--)
    {
        cout << "Осталось выполнить " << i-1 << " циклов\n";
    }
    return 0;
}
```

Программа ForDemo выполняет те же действия, что и ранее рассмотренная While-Demo. Однако вместо изменения переменной loopCount в этом варианте программы введена специальная переменная цикла.

Выполнение цикла начинается с объявления этой переменной i и инициализации ее значением переменной loopCount. Затем проверяется, является ли переменная i положительной. Если да, то программа выводит уменьшенное на 1 значение i и возвращается к началу цикла.

Цикл for также удобен и тогда, когда значение переменной цикла необходимо увеличивать, а не уменьшать. Это осуществляется небольшим изменением цикла for.

```
// ForDemo. Вводится счетчик цикла.
//На экран выводится количество выполненных
// циклов for
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{
     // ввод счетчика цикла
    int loopCount;
    cout << "Введите loopCount: ";
    cin  >> loopCount;

    // вычисляем, пока не нарушится условие
        for (int i = 1; i <= loopCount; i++)
    {
        cout << "Мы завершили " << i << " цикл\n";
    }
    return 0;
}
```



Согласно последнему стандарту языка индексная переменная, объявленная в части инициализации цикла for, известна только в пределах этого цикла. Программисты на C++ в этом случае говорят, что область видимости переменной ограничена циклом for. Например, в инструкции return рассмотренного выше примера, т.е. за пределами цикла, переменная i недоступна и не может использоваться. Однако этого новейшего правила придерживаются далеко не все компиляторы, и вам нужно протестировать свой компилятор, чтобы узнать, как он действует в этом случае.

Вы можете задать вопрос: если цикл for эквивалентен циклу while, зачем забивать себе им голову? Цикл for намного легче для понимания, поскольку главные части любого цикла for (инициализация, условие, увеличение) имеют фиксированное местоположение и соответствуют стандартному формату записи.

Избегайте бесконечных циклов

Бесконечным называют такой цикл, который выполняется вечно. Бесконечный цикл создается каждый раз, когда нельзя нарушить условие выполнения цикла, обычно вследствие какой-то ошибки в коде.

Рассмотрим следующую разновидность рассмотренного ранее цикла:

```
while (loopCount > 0)
{
    cout << "Осталось выполнить" <<loopCount << " циклов\n"
}
```

Программист забыл уменьшить переменную loopCount, как это делалось в ранее рассмотренных циклах. В результате получен счетчик цикла, значение которого никогда не изменяется. При этом условие выполнения цикла будет либо всегда истинно, либо всегда ложно. Выполнение такой программы никогда не закончится естественным путем, т.е., как говорят, программа зацикливается.

Но... ничто не вечно под Луной! В конечном счете правительство сменится, компьютер поломается, Microsoft обанкротится... Или цикл прервется, и вам не о чем будет тревожиться.

Существует гораздо больше способов создания бесконечных циклов, чем показано здесь; но обычно они слишком сложны для того, чтобы приводить их в такой простой книжке.

Специальные операторы циклов

В C++ определены две специальные управляющие программой команды — break и continue. Может случиться, что условие работы цикла нарушится не в начале или в конце, а где-то посередине цикла. Рассмотрим программу, которая суммирует введенные пользователем значения. Цикл прерывается, когда пользователь вводит отрицательное число.

Проблема в том, что программа не сможет выйти из цикла, пока пользователь не введет число, но должна сделать это перед тем, как значение будет добавлено к сумме.

В таких случаях используют определенную в C++ команду break. Она немедленно передает управление в конец текущего цикла. После инструкции break программа будет выполнять инструкцию, следующую сразу же после закрывающей скобки цикла.

Схематически работу команды break можно проиллюстрировать так:

```
while(условие)
{
    if (какое-то другое условие)
    {
        break; //выход из цикла
    }
}       //когда программа встретит break,
        //управление будет передано этой строке
```

Вооружась новой командой break, я реализовал в программе BreakDemo решение проблемы последовательного накопления суммы чисел.

```
// BreakDemo — вводим множество чисел.
//              Суммируем эти числа,
//              пока пользователь не введет
//              отрицательное число
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{
    // введите счетчик цикла
    int accumulator = 0;
    cout << "Эта программа суммирует числа, "
         << "введенные пользователем\n";
    cout << "Выполнение цикла "
         << "заканчивается после "
         << "введения отрицательного числа\n";

    // бесконечный цикл
    for(;;)
    {
        // ввод следующего числа
        int value = 0;
        cout << "Введите следующее число: ";
        cin  >> value;
```

```
        // если оно отрицательно...
        if (value < 0)
        {
            // ...тогда ЕЫХОДИМ из цикла
            break;
        }

        // ...иначе добавляем число к общей сумме
        accumulator = accumulator + value;
    }

    // после выхода из цикла
    // выводим результат суммирования
    cout << "\nОбщая сумма равна "
         « accumulator
         «  "\n";

    return 0;
}
```

После объяснения правил пользователю (что нужно ввести, чтобы выйти из цикла, и что делает эта программа) управление получает нечто подобное бесконечному циклу for. Сначала на экран выводится запрос на введение числа с клавиатуры. Проверить его на соответствие критерию выхода из цикла можно только после того, как оно считано программой. Если введенное число отрицательно, управление переходит к оператору break и программа выходит из цикла. Если же число оказывается положительным, программа пропускает команду break и к накопленной сумме добавляет новое значение. После выхода из цикла программа выводит значение общей суммы всех введенных чисел и завершает работу.



При повторяющемся выполнении операций над переменной в цикле проследите, чтобы инициализация переменной осуществлялась еще до входа в цикл. В нашем случае программа обнулила переменную accumulator перед входом в цикл, в котором к ней добавляются новые числа.

Несколько реже используется команда continue. Столкнувшись с ней, программа немедленно возвращается к началу цикла. Остальные инструкции цикла игнорируются. В следующем фрагменте программы отрицательные числа, которые может ввести пользователь, игнорируются:

```
while (1)
{
    //Ввсд значения
    cout << "Введите значение";
    cin   » inputVal;

    //если число отрицательное
    if (inputVal < 0)
    {
        //выводим сообщение об ошибке
        cout « "Не допускается ввод "
             « "отрицательных чисел\п";
        //возвращаемся к началу цикла
        continue;
    }
    //введено приемлемое значение
}
```

Вложенные команды управления

Возвратимся к проблеме перерисовывания экрана монитора. Конечно, некоторые типы структур циклов можно использовать для прорисовки каждого пикселя линии слева направо (или для евреев и арабов сканирование экрана осуществляется справа налево?). Но как реализовать повторяющееся перерисовывание каждой сканируемой линии сверху вниз? (Для австралийских экранов, наверное, снизу вверх?) Для этого нужно включить цикл, сканирующий пиксели слева направо, внутрь цикла, сканирующего линии сверху вниз.

В случае если команды одного цикла находятся внутри другого, цикл называют *вложенным*. В качестве примера вы можете модифицировать программу BreakDemo в программу, которая накапливает сумму чисел любого количества последовательностей. В этой программе, названной NestedDemo, во внутреннем цикле суммируются числа, введенные с клавиатуры, пока пользователь не введет отрицательное значение. Внешний цикл суммирует полученные суммы значений последовательностей до тех пор, пока накопленная сумма примет отличное от нуля значение.

```
// NestedDemo — вводится последовательность чисел.
//              Числа суммируются, пока пользователь
//              не введет отрицательное число.
//              Этот процесс будет повторяться,
//              пока общая сумма не станет равной 0.
#include <stdio.h>
#include <iostream.h>

int main(ir.t arg, char* pszArgs[])
{
    cout << "Эта программа суммирует "
         << "последовательности чисел.\п Ввод каждой "
         << "последовательности завершается "
         << "вводом отрицательного числа \п"
         << "Чтобы завершить ввод последовательностей, "
         << "нужно ввести\пдва отрицательных числа подряд\n";
    // внешний цикл работает с последовательностями чисел
    int accumulator;
    do
    {
        // начинаем ввод очередной последовательности чисел
        accumulator = 0;
        cout << "\nВведите очередную последовательность\n";

        // бесконечный цикл
        for(;;)
        {
            // введение очередного числа
            int value = 0;
            cout << "Введите очередное число: ";
            cin  >> value;

            // если оно отрицательное...
            if (value < 0}
            {
                // ... выходим из цикла
                break;
            }

            // ...иначе добавляем число к
```

```
            // общей сумме
            accumulator = accumulator + value;
        }

        // Вывод результата вычислений...
        cout << "\nОбщая сумма равна "
            << accumulator
            << "\n";

        // ...если накопленная общая сумма чисел
        // последовательности не равна нулю,
        // начинаем работать со следующей
        // последовательностью
    } while (accumulator != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

Инструкция выбора

Последняя управляющая инструкция эффективна, если существует необходимость выбора при ограниченном количестве возможных вариантов. Она похожа на усложненную инструкцию if, которая вместо проверки одного условия анализирует множество разных возможностей:

```
switch (выражение)
{
    case c1:
        //переходим сюда, если выражение == c1
        break;
    case c2:
        //переходим сюда, если выражение == c2
        break;
    default:
        //если ни одно условие не выполнено,
        // переходим сюда
}
```

Значением выражения должно быть целое число (int, long или char); c1, c2, c3 должны быть константами. Инструкция switch выполняется следующим образом: сначала вычисляется значение выражения, а затем оно сравнивается с константами, указанными после служебного слова case. Если константа соответствует значению выражения, то программа передает управление этой ветви. Если ни один вариант не подходит, выполняется условие default.

Рассмотрим для примера следующий фрагмент программы:

```
cout << "Введите 1, 2 или 3:";
cin  >> choice;

switch (choice)
{
    case 1:
        //обработка случая "1"
        break;
    case 2:
        //обработка случая "2"
        break;
```

```
    case 3:
        //' обработка случая "3"
        break;
    default:
        cout << "Вы ввели не 1, не 2 и не 3\п"
}
```

Еще раз напомню, что инструкция switch эквивалентна усложненной инструкции if (с вложенными if-инструкциями); однако, если рассматривается более двух-трех случаев, структура switch оказывается нагляднее.



Для выхода из инструкции switch необходимо использовать команды break, иначе управление будет переходить от одного случая к следующему.

Часть II

Становимся программистами



...я и говорю: "Официант! Официант! У меня в супе — жучок!". А он мне отвечает: "Извините, сэр, но наш шеф-повар занялся программированием..."

В этой части...

*Выполнять операции **сложения** или умножения (и даже логические операции) — это одно, а писать настоящие программы — это нечто совсем иное. В **этой** части рассматривается все необходимое для того, чтобы стать настоящим программистом.*

Глава 6

Создание функций

В этой главе...

- ✓ Написание и использование функций
- ✓ Подробный анализ функций
- ✓ Перегрузка функций
- ✓ Определение прототипов функций
- ✓ Хранение переменных в памяти

Очень часто при написании программ возникает необходимость разделить большую программу на меньшие части, отлаживать которые намного легче. Программы из предыдущих глав слишком малы, чтобы можно было по-настоящему оценить пользу такого разделения. Но реальные программы из больших проектов состоят из тысяч (и даже миллионов!) строк. Поэтому большие программы просто невозможно написать, не разбивая их на отдельные модули.

С++ позволяет разделить код программ на части, называемые функциями. Сами функции могут быть записаны и отлажены отдельно от остального кода программы.



Хорошая функция может быть описана одним предложением с минимальным количеством слов "и" и "или". Например, функция `sumSequence()` суммирует последовательность целочисленных значений, введенных пользователем. Это определение весьма компактно и легко воспринимается.

Возможность разбивать программу на части с последующей отладкой каждой функции в отдельности существенно снижает сложность создания больших программ. Этот подход является, по сути, простейшей формой инкапсуляции (смотрите главу 12, "Знакомство с объектно-ориентированным программированием", где вопросы инкапсуляции рассматриваются подробнее.)

Написание и использование функций

Функции лучше всего изучать на примерах. Эта часть начинается с программы `FunctionDemo`, которая показывает, как упростить рассмотренную в главе 5 программу `NestDemo`, определив дополнительную функцию. На примере программы `FunctionDemo` я постараюсь объяснить, как определять и использовать функции. Эта программа будет служить образцом для их дальнейшего изучения.

`NestDemo` содержит два цикла. Во внутреннем цикле суммируется последовательность введенных пользователем чисел. Он включен во внешний цикл, который повторяет процесс, пока пользователь не изъявит желания его прекратить. Разделение этих двух циклов делает программу более наглядной.

В программе `FunctionDemo` показано, как упростить программу `NestDemo` с помощью создания функции sumSequence {] .



Согласно синтаксису C++ справа от имени функции должны присутствовать две круглые скобки. В них обычно указываются параметры функций.

```
// FunctionDemo — демонстрация использования
//                функций. Внутренний цикл
//                программы оформлен как
//                отдельная функция

#include <stdio.h>
#include <iostream.h>

// sumSequence — суммирует последовательность
//               чисел, введенных с клавиатуры, пока
//               пользователь не введет отрицательное
//               число. Возвращает сумму введенных чисел
int sumSequence(void)
{
    // бесконечный цикл
    int accumulator = 0;
    for (;;)
    {
        // ввод следующего числа
        int value = 0;
        cout << "Введите следующее число: ";
        cin  >> value;

        // если оно отрицательное...
        if (value < 0)
        {
            // ...тогда выходим из цикла
            break;
        }

        // ...иначе добавляем число к
        // переменной accumulator
        accumulator = accumulator + value;
    }

    // возвращаем значение суммы
    return accumulator;
}

int main(int arg, char* pszArgs[])
f
    cout << "Эта программа суммирует последовательности\n"
         << "чисел. Каждая последовательность\n"
         << "заканчивается отрицательным числом.\п"
         << "Ввод серий завершается вводом \п"
         << "двух отрицательных чисел подряд \п";

    // суммируем последовательности чисел...
    int accumulatedValue;
    do
    {
        // суммируем последовательности чисел,
        // введенных с клавиатуры
        cout << "\nВведите следующую последовательность\n";
```

```
        accumulatedValue = sumSequence();

        // вывод общей суммы на экран
        cout << "\nОбщая сумма равна "
             << accumulatedValue
             << "\n";

    // ...пока возвращаемая сумма не равна 0
    } while (accumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

Вызов функции sumSequence ()

Главная часть программы сконцентрирована в фигурных скобках, следующих после объявления функции main(). Эта часть кода очень напоминает программу NestDemo.

Различие состоит в том, что внутри функции main() содержится выражение accumulatedValue = sumSequence (). В правой части этого выражения вызывается функция sumSequence (). Возвращаемое функцией значение сохраняется в переменной accumulatedValue, а затем выводится на экран. Главная программа выполняет цикл до тех пор, пока значение суммы, возвращаемой внутренней функцией, остается отличным от 0. Нулевое значение говорит о том, что пользователь закончил вычисление сумм последовательностей.

Вызвать функцию — это значит начать выполнение кода, который содержится в ней. После завершения этого процесса программа передаст управление инструкции, следующей непосредственно за вызовом функции.

Определение функции sumSequence ()

Определение этой функции начинается с инструкции int sumSequence (void). Заключенный в фигурные скобки блок кода называется *телом функции.* Как видите, тело функции sumSequence () идентично внутреннему циклу программы NestDemo.

Работа программы осуществляется следующим образом: функция main выполняет цикл, внутри которого (там, где в программе NestDemo находился внутренний цикл) происходит вызов функции sumSequence (). Она вычисляет сумму, а результат возвращает в функцию main программы, после чего продолжается выполнение внешнего цикла.

Подробный анализ функций

Функции являются первоосновой программ C++. Поэтому каждый программист должен отчетливо понимать все нюансы их определения, написания и отладки.

Функцией называют логически обособленный блок кода C++, имеющий следующий вид:

```
<тип возвращаемого значения> name(<аргументы функцию)
{

    return<выражение>;
}
```

Аргументами функции называются значения, которые можно передать ей при вызове. В *возвращаемом значении* указывается результат, который функция возвращает по окончании работы. Например, в вызове функции возведения в квадрат square (10) 10 является аргументом, а возвращаемое значение равно 100.

И аргументы, и возвращаемое значение функции необязательны. Если что-то отсутствует, вместо него используется ключевое слово void. Значит, если вместо списка аргументов используется void, то при вызове функция не получает никаких аргументов (как и в рассмотренной программе FunctionDemo). Если же возвращаемый тип функции — void, то вызывающая программа не получает от функции никакого значения.

В программе FunctionDemo используется функция с именем sumSequence (), которая не имеет аргументов и возвращает значение целочисленного типа.



Тип аргументов функции по умолчанию — void, поэтому функцию int fn(void) можно записать как int fn().

Использование функции позволяет работать с каждой из двух частей программы FunctionDemo в отдельности. При написании функции sumSequence () я концентрирую внимание на вычислении суммы чисел и не думаю об остальном коде, вызывающем эту функцию.

При написании функции main() я работаю с возвращаемой функцией sumSequence () — суммой последовательности (на этом уровне абстракции я знаю только, что выполняет функция, а не как именно она это делает).

Простые функции

Функция sumSequence () возвращает целое значение. Функции могут возвращать значение любого стандартного типа, например double или char (типы переменных рассматриваются в главе 5, "Операторы управления программой").

Если функция ничего не возвращает, то возвращаемый тип помечается как void.



Функции различаются по типу возвращаемого значения. Так, *целочисленной функцией* называют ту, которая возвращает целое значение. Функция, которая ничего не возвращает, известна как *void-функция.* Далее приведен пример функции, выполняющей некоторые действия, но не возвращающей никаких значений.

```
void echoSquare(>
{
    cout << "Введите значение:";
    cin >> value;
    cout << "\n"«value*value « "\n";
    return;
)
```

Сначала управление передается первой инструкции после открывающей скобки, затем поочередно выполняются все команды до инструкции return (которая в данном случае не требует аргумента). Эта инструкция передает управление вызывающей функции.



Инструкция return в void-функциях является необязательной. Если она отсутствует, то выполнение функции прекращается при достижении закрывающей фигурной скобки.

Функции с аргументами

Функции без аргументов используются редко, так как связь с такими функциями односторонняя, т.е. осуществляется только посредством возвращаемых значений. Аргументы функций позволяют установить двустороннюю связь — через передаваемые параметры и возвращаемые значения.

Функции с одним аргументом

Аргументами функции называют значения, которые передаются функции во время вызова. В следующем примере определяется и используется функция square{>, которая возвращает квадрат переданного в качестве аргумента числа типа double:

```
// SquareDemo — демонстрирует использование
// функции, обрабатывающей аргументы

#include <stdio.h>
#include <iostream.h>

// square - возвращает квадрат аргумента
//          doubleVar — введенное значение
//          returns — квадрат doubleVar
double square(double doubleVar)
{
    return doubleVar * doubleVar;
}

// sumSequence — суммирует последовательность
// чисел, зведенных с клавиатуры и возведенных в
// квадрат, пока пользователь не введет
//отрицательное число.
// Возвращает сумму квадратов введенных чисел

int sumSequence(void)
{
    // бесконечный цикл
    int accumulator= 0;
    for(;;)
    {
        // введение следующего числа
        double dValue = 0;
        cout << "Введите следующее число:  ";
        cin  >> dValue;

        // если оно отрицательное...

        if (dValue < 0)
        {
            // . . .то выходим из цикла
            break;
        }

        // ...иначе вычисляем квадрат числа
        int value = (int)square(dValue);

        // теперь добавляем квадрат к
        // accumulator
        accumulator= accumulator + value;
    }

    // возвращаем накопленное число
    return accumulator;
}

int main(int arg, char* pszArgs[])
{
```

```
    cout « "Эта программа суммирует\n"
        << "множественные серии чисел. \n"
        << "Введение каждой последовательности\n"
        << "заканчивается введением\п"
        << "отрицательного числа.\n"
        « "Серии последовательностей вводятся\n"
        << "до тех пор, пока не встретятся\n"
        << "два отрицательных числа\n";

    // Продолжаем суммировать числа...
    int accumulatedValue;
    do
    {
        // суммируем последовательность чисел,
        // введенных с клавиатуры
        cout << "\пВведите следующую последовательность\n";
        accumulatedValue = sumSequence();

        // теперь выводим результат суммирования
        cout << "\nОбщая сумма равна "
             << accumulatedValue
             « "\п";

    // ...пока возвращаемая сумма не равна О
    } while (accumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

По сути, перед вами все та же программа FunctionDemo, но теперь она суммирует квадраты введенных чисел. В функции square() играющее роль параметра число возводится в квадрат. Проведены незначительные изменения и в функции sumSequence (): если раньше мы суммировали введенные числа, то теперь суммируем значения, возвращаемые функцией square ().

Функции со многими аргументами

Функции могут иметь не один аргумент. В этом случае аргументы разделяются запятыми. Например, следующая функция возвращает произведение двух аргументов:

```
int product(int arg1, int arg2)
{
    return arg1*arg2;
}
```

Преобразование типов

В программе содержится оператор, с которым вы раньше не встречались:

```
accumulator = accumulator + (int)dValue;
```

Префикс (int) перед dValue означает, что программист перед выполнением суммирования преобразует переменную dValue из ее текущего типа (в данном случае double) в int. Такая форма записи позволяет преобразовывать значения одного типа к другому.

С++ допускает преобразование любых числовых величин в числовые величины другого типа. Компилятор С++ всегда выполняет приведение типов и без вашего вмешательства, но в таком случае для проверки правильности приведения он должен выдавать об этом соответствующее предупреждение. Поэтому во избежание ошибок рекомендуется использовать явное преобразование типов.

Функция main()

Служебное слово main() в нашей стандартной программе означает не что иное, как функцию (возможно, с необычными для вас аргументами и не требующую прототипа).

При компиляции программы С++ добавляет некоторый стандартный программный код, выполняемый до того, как начинает выполняться функция main (). Этот код настраивает программную среду, в которой выполняется ваша программа, например открывает потоки ввода и вывода по умолчанию.

После настройки среды выполнения этот код вызывает функцию main(), и лишь тогда происходит выполнение операторов программы. При завершении программа выходит из функции main (), после чего вторая часть стандартного кода С++ освобождает занятые программой системные ресурсы и передает управление операционной системе.

Перегрузка функций

С++ позволяет программистам называть несколько разных функций одним и тем же именем Эта возможность называется перефузкой функций (function overloading).

Вообще говоря, две функции в одной программе не могут иметь одинаковых имен, так как компилятор С++ просто не сможет их различить.

Однако используемое компилятором внутреннее имя функции включает число и типы аргументов (но не возвращаемое значение), и поэтому следующие функции являются разными:

```
void someFunction(void)
{
    //...выполнение некоторой функции
}
void someFunction(int n)
(
    //... выполнение другой функции
}
void someFunction(double d)
i
    //... выполнение еще одной функции
}
void someFunction(int n1, int n2)
{
    //... выполнение еще одной функции,
    // отличной от предыдущих
)
```

Компилятор С++ знает, что функции someFunction (void) , someFunction(int), someFunction(double), someFunction(int, int) не одинаковы. В мире компьютеров встречается довольно много таких вещей, для которых можно найти аналогии в реальном мире. Существуют они и для перегрузки функций.

Вы знаете, что тип аргумента *void* указывать не обязательно. Например, SumFunction(void) и SumFunction() вызывают одну и ту же функцию. Фактически функция может иметь сокращенное название, в нашем случае someFunction (), так же как и меня, можно называть просто Стефаном. Если бы Стефанов больше нигде не было, меня всегда могли бы называть только по имени. Но в действительности нас побольше, и если кому-то посчастливится попасть в общество нескольких Стефанов, то к ним придется обращаться по имени и фамилии (кстати, напомню, что меня зовут Стефан Дэвис). Пользуясь полным именем, никто не ошибется, даже если Стефанов будет несколько. Поэтому, пока полные имена функций в профамме уникальны, конфликты между ними невозможны.

Такие аналогии между компьютерным и реальным миром не должны вас удивлять, так как компьютерный мир создан людьми.

Типичное приложение может выглядеть следующим образом:

```
int intVariablel, intVariable2;
double doubleVariable;

//функции различаются по типу передаваемых аргументов
someFunction();                //вызов someFunction(void)
someFunction(intVariablel);    //вызов someFunction(int)
someFunction(doubleVariable); //вызов someFunction(double)
someFunction(intVariablel, intVariable2);
                               //вызов someFunction(int, int)

// с константами функции работают аналогично
someFunction(1);                // вызов someFunction(int)
someFunction(1.0);              // вызов someFunction(double)
someFunction(1,2);              // вызов someFunction(int, int)
```

В каждом случае типы аргументов соответствуют тем, которые значатся в полном имени каждой функции.



Тип возвращаемого значения в полное имя функции (называемое также ее сигнатурой) не входит. Следующие две функции имеют одинаковые имена (сигнатуры) и поэтому не могут использоваться в одной программе:

```
int someFunction(int n);
//полным именем этой функции является someFunction(int)
double someFunction(int n); //имеет то же полное имя
```



Следующий код вполне допустим:

```
int someFunction(int n);
double d = someFunction(10),•
//преобразуем тип полученного значения
```

В этом фрагменте возвращаемые функцией значения типа int преобразуются в double. Но следующий код некорректен:

```
int someFunction(int n);
double someFunction(int n);
double d = someFunction(10);
//В этом случае мы преобразуем тип полученного
//целочисленного значения или используем вторую функцию?
```

В этом случае С++ не поймет, какое значение он должен использовать — возвращаемое double-функцией или ее целочисленным вариантом.

Определение прототипов функций

Как уже отмечалось, любой фрагмент кода программист может оформить как функцию, присвоив ей полное имя, таким образом объявляя ее для дальнейшего использования.

Функции sumSequence () и square O, с которыми вы встречались в этой главе, были определены до того, как вызывались. Но это не означает, что нужно всегда придерживаться именно такого порядка. Функция может быть определена в любой части модуля (модуль — это другое название исходного файла С++).

Однако должен использоваться какой-то механизм, уведомляющий функцию main() о функциях, которые могут вызываться ею. Рассмотрим следующий код:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double arg1, int arg2)
{
    //...выполнение каких-то действий
}
```

При вызове функции someFunc() внутри main() полное ее имя неизвестно. Можно предположить, что именем функции является someFunc (int, int) и возвращаемое ею значение имеет тип void. Однако, как видите, это вовсе не так.

Согласен, компилятор C++ мог бы быть не таким ленивым и просмотреть весь модуль для определения сигнатуры функции. Но он этого не сделает, и с этим прихо" дится считаться[13]. Таков мир: любишь кататься — люби и саночки возить.

Поэтому нам нужно проинформировать main() о полном имени вызываемой в ней функции до обращения к ней. Для этого используют прототипы функций.

Прототип функции содержит ее полное имя с указанием типа возвращаемого значения. Использование прототипов рассмотрим на следующем примере:

```
int someFunc(double, int);
int main(int argc, char* pArgs[])
(
    someFunc(1, 2);
|
int someFunc(double arg1, int arg2)
{
    //...выполнение каких-то действий
}
```

Использованный прототип объясняет миру (по крайней мере той его части, которая следует после этого объявления), что полным именем функции someFunc () является someFunc(double, int). Теперь при ее вызове в main() компилятор поймет, что 1 нужно преобразовать к типу double. Кроме того, функция main () осведомлена, что someFunc() возвращает целое значение.



Вызов функции, возвращающей значение, является выражением. Следовательно, с ним можно работать так же, как и с любым другим значением этого же типа.

Хранение переменных в памяти

В памяти переменные хранятся в трех разных местах. Переменные, объявленные внутри функции, называются локальными. В следующем примере переменная localVariable является локальной по отношению к функции fn ():

```
int globalVariable;
void fn()
{
```

---

13 *Более того, как вы узнаете позже, тела функции в данном модуле может и не оказаться.* — Прим. ред.

```
    int localVariable;
    static int staticVariable;
}
```

До вызова fn() переменной localVariable не существует. После окончания работы функции она оставляет этот бренный мир и ее содержимое навсегда теряется. Добавлю, что доступ к ней имеет только функция fn(), остальные использовать ее не могут.

А вот переменная globalVariable существует на протяжении работы всей программы и в любой момент доступна всем функциям.

Статическая переменная staticVariable является чем-то средним между локальной и глобальной переменными. Она создается, когда программа при выполнении достигает описания переменной (грубо говоря, когда происходит первый вызов функции). К тому же staticVariable доступна только из функции fn(). Но, в отличие от localVariable, переменная staticVariable продолжает существовать и после окончания работы функции. Если в функции fn() переменной staticVariable присваивается какое-то значение, то оно сохранится до следующего вызова fn().

Глава 7

Хранение последовательностей в массивах

В этой главе...

- ✓ Преимущества массивов
- ✓ Работа с массивами
- ✓ Использование символьных массивов
- ✓ Управление строками
- ✓ Устранение устаревших функций вывода

Массивом называется последовательность переменных одного типа, использующая одно имя; для ссылки на конкретное значение в ней используется индекс. Массивы удобны для хранения огромного количества взаимосвязанных значений. Например, голы, забитые каждым игроком футбольной команды, естественно сохранять именно в массивах. В C++ допускаются и многомерные массивы. Например, массивы с количеством голов можно сохранить в массиве месяцев — это позволит работать с количеством голов, забитых каждым игроком в определенном месяце.

Из этой главы вы узнаете, как инициализировать и использовать массивы не только для работы, но и для развлечения. А еще я расскажу об очень полезном виде массивов — строках, которые в C++ являются массивом значений типа char.

Преимущества массивов

Рассмотрим следующую проблему. Вам нужна программа, которая сможет считывать последовательность чисел, введенных с клавиатуры. Будем использовать уже привычное правило, согласно которому ввод чисел завершается после первого отрицательного значения. Программа один раз считывает числа и только после того, как все числа прочитаны, отображает их на стандартном устройстве вывода.

Можно попытаться хранить числа в независимых переменных, например:

```
cin >> value1;
if (value1 >= 0)
{
    cin >> value2;
    if (value2 >= 0)
    {
        ...
```

Однако нетрудно заметить, что этот подход позволит управлять последовательностью, которая будет состоять всего лишь из нескольких чисел, а кроме того, такая запись выглядит довольно уродливо. В нашем случае нужна такая структура данных, которая, как и любая переменная, имеет свое имя, но может содержать больше одной переменной. Для этого как раз и используются массивы.

С помощью массивов можно легко решить проблему работы с подобными последовательностями. В приведенном далее фрагменте объявляется массив valueArray, в котором можно хранить до 128 целых значений. Затем он заполняется числами, введенными с клавиатуры.

```
int value;

// объявление массива, способного содержать
// до 128 чисел типа int
int valueArray[128];

//определение индекса, используемого для
//доступа к элементам массива; его значение
//не должно превышать 128
for (int i = 0; i < 128; i++)
{
    cin >> value;
    //выходим из цикла, если пользователь
    //вводит отрицательное число
    if (value < 0) break;
    valueArray[i] = value;
)
```

Во второй строке кода (без учета комментариев) объявлен массив valueArray. Первым в объявлении указывается тип элементов массива (в нашем случае это int), за ним следует имя массива, последним элементом являются открывающая и закрывающая квадратные скобки, в которых записывается максимальное число элементов массива. В нашем случае массив valueArray может содержать до 128 целочисленных значений.

Компьютер считывает число с клавиатуры и сохраняет его в следующем элементе массива valueArray. Доступ к элементам массива обеспечивается с помощью имени массива и индекса, указанного в квадратных скобках. Первое целое массива обозначается как valueArray[0], второе — как valueArray[1] И т.д.

Запись valueArray[i] представляет собой i-й элемент массива. Индексная переменная i должна быть перечислимой, т.е. ее типом может быть char, int или long. Если valueArray является массивом целых чисел, то элемент valueArray[i] имеет тип int.

Работа с массивами

В представленной ниже программе осуществляется ввод последовательности целых чисел (до первого отрицательного числа), затем эта последовательность и сумма ее элементов выводятся на экран.

```
// ArrayDemo — демонстрирует использование
// массивов. Считывает последовательность
// целых чисел и отображает их по порядку
#include <stdio.h>
#include <iostream.h>

// объявления прототипов функций
int  sumArray(int integerArray[], int sizeOfloatArray);
void displayArray(int integerArray[], int sizeOfloatArray);

int main(int nArg, char* pszArgs[]}
{
     // Описываем счетчик цикла
    int nAccumulator = 0;
    cout << "Эта программа суммирует числа,"
```

```
         << " введенные пользователем\n";
    cout << "Цикл прерывается, когда"
         << " пользователь вводит"
         << " отрицательное число\п";

    // сохраняем числа в массиве
    int inputValues[128];
    int numberOfValues = 0;
    for(; numberOfValues < 128; numberOfValues++)
    {
        // ввод очередного числа
        int integerValue;
        cout << "Введите следующее число: ";
        cin  >> integerValue;

        // если оно отрицательное...
        if {integerValue < 0)
        {
            // ...тогда выходим из цикла
            break;
        }

        // ...иначе сохраняем число в массиве
         inputValues[numberOfValues] = integerValue;
    }

    // теперь выводим значения и их сумму
    displayArray(inputValues, numberOfValues);
    cout << "Сумма введенных чисел равна "
         << sumArray(inputValues, numberOfValues)
         «  "\п";
    return 0;
}

// displayArray — отображает элементы массива
// integerArray длиной sizeOfloatArray
void displayArray(int integerArray[], int sizeOfArray)
{
    cout << "В массиве хранятся"
         << " следующие значения:\п";
    for (int i = 0; i < sizeOfArray; i++)
    {
        cout.width(3);
        cout « i « ":  " « integerArray[i] << "\n";
    }
    cout << "\n";
}

// sumArray — возвращает сумму элементов
// целочисленного массива
int sumArray(int integerArray[], int sizeOfArray)
{
    int accumulator = 0;
    for (int i = 0; i < sizeOfArray; i++)
    {
        accumulator += integerArray[i];
    }
    return accumulator;
}
```

Программа ArrayDemo начинается с объявления прототипов функций sumArray() и displayArray(), которые понадобятся нам позже. Главная часть программы содержит довольно скучный цикл ввода значений. На этот раз вводимые значения сохраняются в массиве inputvalues.

Ввод осуществляется в цикле for. Введенное значение сначала сохраняется в локальной переменной integerValue. Если оно оказывается отрицательным, оператор break прерывает выполнение цикла. Если же нет, значение integerValue копируется в массив.

Целочисленная переменная numberOfValues используется в качестве индекса массива. Она инициализирована нулем еще до начала цикла for. При каждой итерации индекс увеличивается. В условии выполнения цикла for осуществляется контроль за тем, чтобы количество введенных чисел не превышало 128, т.е. размера массива (после введения 128 чисел программа переходит к выводу элементов массива на экран независимо от того, ввел пользователь отрицательное число или нет).



В объявлении массива inputvalues было указано, что его максимальная длина равна 128. При записи большего числа данных, чем определено в объявлении, ваша программа может работать неправильно и даже аварийно завершать работу. Поэтому лучше застраховаться и оставить больше места для хранения данных. Неважно, насколько велик массив; всегда нужно следить за тем, чтобы операции с массивом не приводили к выходу за его пределы.

Функция main заканчивается выводом на экран содержимого массива и суммы его элементов. Функция displayArray() содержит обычный цикл for, который используется для прохождения по массиву. Каждый очередной элемент массива добавляется к переменной accumulator. Передаваемый функции параметр sizeOfArray включает количество значений, содержащихся в массиве.

Напомню еще раз, что индекс массива в C++ отсчитывается от 0, а не от 1. Кроме того, обратите внимание, что цикл for прерывается в тот момент, когда значение i становится равным sizeOfArray. Вы же не хотите добавлять все 128 элементов массива integerArray к accumulator? Ни один элемент массива, индекс которого больше или равен числу sizeOfArray, учитываться не будет.

Инициализация массива

Локальная переменная нежизнеспособна до тех пор, пока ей не присвоят значение. Другими словами, пока вы в ней что-то не сохраните, она будет содержать мусор. Локальное описание массива происходит так же: пока каждому элементу не присвоят какие-либо значения, в ячейках массива будет содержаться мусор. Локальную переменную следует инициализировать при ее объявлении, и еще в большей степени это справедливо для массивов. Слишком уж легко наткнуться на неработоспособную ячейку в неинициализированном массиве.

К счастью, массив может быть инициализирован сразу во время объявления, например:

```
float floatArray[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

В этом фрагменте элементу floatArray[0] присваивается значение 0, floatArray[1] — 1, floatArray[2] — 2 и т.д.

Размер массива может определяться и количеством инициализирующих констант. Например, перечислив в скобках значения инициализаторов, можно ограничить размер массива floatArray пятью элементами. C++ умеет очень хорошо считать (по крайней мере, его можно спокойно использовать для этого). Так, следующее объявление идентично представленному выше:

```
float floatArray[] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Выход за границы массива

Математики пересчитывают содержимое массивов, начиная с элемента номер 1. Первым элементом математического массива x является x(1). Во многих языках программирования также начинают перечисление элементов массива с 1. Но в C++ массивы индексируются начиная с 0! Первый элемент массива C++ обозначается как valueArray[0]. Первый индекс массива C++ нулевой; поэтому последним элементом 128-элементного целочисленного массива является integerArray[127], а не integerArray [128].

К сожалению, в C++ не проверяется выход индекса за пределы массива. Этот язык будет рад предоставить вам доступ к элементу integerArray[200]. Более того, C++ позволит вам обратиться даже к integerArray[-15].

Приведем такую аналогию. Длину трассы можно измерять равными промежутками между многочисленными придорожными столбами. Назовем их единицами измерения длины дороги. Чтобы попасть к моему дому, нужно свернуть с главной трассы и по прямой ехать до конца дороги, длина которой равна точно девяти столбам. Если начинать отсчет с последнего телеграфного столба на трассе, тогда столб у моего дома будет иметь 10-й номер.

Отсчитывая столбы на трассе, можно получить доступ к любой позиции на дороге. Если отмерять расстояние от трассы до трассы, то получим, что оно равно нулю столбов. Следующей дискретной точкой отсчета является первый столб и т.д., пока вы не доберетесь до моего дома, который находится на расстоянии девяти столбов.

Конечно, вы можете отмерить расстояние в 20 столбов от трассы. Но дороги там уже не будет (помните, она кончается у моего дома?). Трудно точно сказать, что вы сможете там найти. Вы можете оказаться на следующей трассе, где-нибудь в поле или даже в чьей-то спальне (вот будет весело!). Угадать, куда в этом случае попадешь, довольно трудно, но хранить там что-то еще рискованнее. Оставить вещь в поле — это одно, но забросить ее в комнату моего соседа — совсем другое (я-то знаю это наверняка, так как каждый раз, когда газеты исчезают из моего почтового яшика, они оказываются в гостиной моего соседа).

То же происходит и в программировании. Пытаясь прочитать 20-й элемент (array[20]) 10-элементного массива, вы получите то или другое случайное число (а возможно, программа просто аварийно завершит работу). А записывая в array[20] какое-то значение, вы получите непредсказуемый результат. Скорее всего, ничего в него не запишется, но возможны и другие странные реакции вплоть до краха программы.



Самой распространенной ошибкой является неправильное обращение к последнему элементу по адресу integerArray 1128]. Хотя это всего лишь следующий за концом массива элемент, записывать или считывать его не менее опасно, чем любой другой некорректной адрес.

Использовать ли массивы

Разумеется, программа ArrayDemo делает то же самое, что и не основанные на массивах программы, которые рассматривались раньше. Правда, в этой версии несколько изменен ввод множества чисел, но вы вряд ли будете потрясены этой особенностью.

И все же в возможности повторного отображения введенных значений кроется значительное преимущество использования массивов. Массивы позволяют программе многократно обрабатывать серии чисел. Главная программа была способна передать массив входных значений функции displayArray () для отображения, а затем в SumArray() для суммирования.

Определение и использование массивов

Массивы представляют собой весьма удобную структуру для хранения последовательности чисел. В некоторых приложениях приходится работать с последовательностью последовательностей. Классическим примером такой матричной конфигурации является крупноформатная таблица, распланированная по образцу шахматной доски (каждый ее элемент имеет две координаты — *x* и *y*).

В С++ матрицы определяются следующим образом:

```
int intMatrix[10][5];
```

Эта матрица может иметь 10 элементов в одном измерении и 5 в другом, что в сумме составляет 50 элементов. Другими словами, intMatrix является 10-элементным массивом, каждый элемент которого — массив из 5 элементов. Легко догадаться, что один угол матрицы обозначается intMatrix[0] [0], тогда как второй — intMatrix[9][4].

Индексы intMatrix можно рассматривать в любом удобном порядке. По какой оси отложить длину 10 — решайте сами, исходя из удобства представления. Матрицу можно инициализировать так же, как и массив:

```
int intMatrix[2][3] = {{1, 2, 3), (4, 5, 6}};
```

Здесь фактически выполняется инициализация двух трехэлементных массивов: intMatrix [0] — значениями I, 2 и 3, а intMatrix [1] — 4, 5 и 6 соответственно.

Использование символьных массивов

Элементы массива могут быть любого типа. В С++ возможны массивы любых числовых типов — float, double, long, однако символьные массивы имеют особое значение.

Слова разговорной речи могут быть интерпретированы как массивы символов. Массив символов, содержащий мое имя, таков:

```
char sMyNarae[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
```

Мое имя можно отобразить с помощью следующей небольшой программы:

```
// CharDisplay — выводит на экран массив
// символов в окне MS DOS
#include <stdio.h>
#include <iostream.h>

// объявления прототипов
void displayCharArray(char stringArray[],
                      int sizeOfloatArray);

int main(int nArg, char* pszArgs[])
{
    char charMyName[] = { 'S', 't', 'e', 'p', 'h', 'e', 'n'};
    displayCharArray(charMyName, 7);
    cout << "\n";
    return 0;
}

//displayCharArray — отображает массив символов,
// по одному при каждой итерации
void displayCharArray(char stringArray[],
                      int sizeOfloatArray)
{
```

```
    for( int i = 0; i<sizeOfloatArray; i++)
    {
        cout << stringArray[i];
    }
}
```

В программе объявлен фиксированный массив символов, содержащий, как вы могли заметить, мое имя (какое еще может быть лучше?). Этот массив передается в функцию displayCharArray() вместе с его длиной. Функция displayCharArray() идентична функции displayArray() из нашего предыдущего примера, но в этом варианте вместо целых чисел ею выводятся символы.

Программа работает довольно хорошо; но одно неудобство все-таки есть: всякий раз вместе с самим массивом необходимо передавать его длину. Однако можно придумать правило, которое поможет решить нашу проблему. Если бы мы знали, что в конце массива находится специальный кодовый символ, то не потребовалось бы передавать размеры массива.

Будем использовать для указания конца массива нулевой символ.

Нулевой символ и обычный 0, вводимый с клавиатуры, — разные понятия. Кодом обыкновенного символа 0 является 0x30. Нуль-символ — это символ, каждый бит которого равен 0 (т.е. 0x00). Для того чтобы этот символ было легче отличить от 0, его часто записывают как \0.



Символ \y является символом, числовое значение которого равно y. Изменим предыдущую программу, используя это правило:

```
// DisplayString — выводит на экран массив
// символов в окне MS DOS
#include <stdio.h>
#include <iostream.h>

// объявления прототипов
void displayString(char stringArray[]);

int main(int nArg, char* pszArgs[])
{
    char charMyName[] =
            {'S', 't', 'e', 'p', 'h', 'e', 'n', 0};
    displayString(charMyName);
    cout << "\n";
    return 0;
}

// displayString — посимвольно выводит на экран строку
void displayString(char stringArray[])
{
    for(int i = 0; stringArray[i] != 0; i++)
    {
        cout << stringArray[i];
    }
}
```

Массив charMyName объявляется как массив символов с дополнительным нулевым символом (\0) в конце. Программа displayString итеративно проходит по символьному массиву, пока не встретит нуль-символ.

Поскольку в функции displayString() больше нет необходимости передавать куда-либо длину символьного массива, использовать ее проще, чем displayCharArray().

К тому же displayString() работает даже тогда, когда размер строки символов во время компиляции неизвестен. А это случается намного чаще, чем вы думаете (смотрите главу 6, "Создание функций").

Включать нулевой символ в символьные массивы очень удобно, и в языке C++ он используется повсеместно. Для таких массивов даже придумали специальное имя.



Строка — это символьный массив с завершающим нулевым символом.

Инициализировать строку в C++ можно с использованием двойных кавычек. Этот способ более удобен, чем тот, в котором используются одинарные кавычки для каждого символа. Строки 8 и 9 из предыдущего примера можно записать более компактно:

```
char szMyName[] = "Stephen";
```

Соглашение об использовании имен для обозначения строк с завершающим нулем использует префикс sz. Такая запись является соглашением и не более.



Строка "Stephen" содержит восемь, а не семь символов — не забывайте о нулевом символе!

Управление строками

Программистам часто приходится работать со строками. В C++ для этого можно использовать стандартные библиотечные функции обработки строк, что существенно облегчает жизнь. Чтобы понять, как они работают, попытаемся самостоятельно написать что-то подобное.

Написание функции, соединяющей две строки

Вы можете написать собственный вариант обрабатывающей строки функции. Для этого нужно использовать семантику массива и добавить проверку наличия в конце массива нулевого символа. Рассмотрим следующий пример:

```
// Concatenate — объединение двух строк, которые
// разделяются символом " — "
#include <stdio.h>
#include <iostream.h>

// Включаем файл, необходимый для использования
// функций работы со строками
#include <string.h>

// Объявления прототипов
void concatString(char szTarget[], char szSource[]);

int rnain(int nArg, char* pszArgs[])
i
    // считываем первую строку...
    char szString1[256];
    cout << "Введите строку #1: ";
    cin.getline(szString1, 128);

    // ...теперь вторую...
    char szString2[128];
```

```
    cout << "Введите строку #2: ";
    cin.getline(szString2, 128);

    // ...присоединим " — " к первой строке...
    concatString(szString1, " — ");
    // strcat(szString1, " - " ) ;

    // ...теперь добавим вторую строку...
    concatString(szString1, szString2);
    // strcat(szString1, szString2);

    // ...и выведем результат на экран
    cout << "\n" << szStringl << "\n";

    return 0;
}

// concatString ~ присоединяет строку szSource
//                к концу строки szTarget
void concatString(char szTarget[], char szSource[])
{
    // находим конец первой строки
    int targetIndex = 0;
    while (szTarget[targetIndex])
    {
        targetIndex++;
    }

    // присоединяем вторую строку к концу первой
    int sourceIndex = 0;
    while (szSource [sourceIndex] )
    {
        szTarget[targetIndex] =
            szSource[sourceIndex];
        targetIndex++;
        sourceIndex++;
    }

    // вписываем в конец ноль-символ
    szTarget[targetIndex] = '\0';
}
```

Функция main **считывает две строки, используя функцию** `getline{}`. **(При использовании варианта** `cin >> szString` **информация считывается до первого пробела, в нашем же случае строка считывается до символа начала новой строки.)**

Функция main () объединяет две строки, используя `concatString{}` **перед выводом результатов. Функция** `concatString()` **присоединяет второй аргумент** (`szSource`) **к концу** первого (`szTarget`). **Эта задача решается в несколько этапов.**

В первом цикле проводится поиск конца строки `szTarget`. **Функция** `concatString;` **итеративно проходит по строке** `szTarget`, **пока значение** `targetlndex` **не станет равным индексу нулевого символа в конце строки. В этот момент переменная** `targetIndex` **указывает на последний символ в строке** `szTarget`.



Цикл `while(value!=0)` **можно записать короче:** `while(value)`, **так как рассматриваемое условие принимает значение** `false`, **если** `value` **равно 0, и** `true` **в прочих случаях.**

Во втором цикле мы итеративно проходим по строке szSource, копируя каждый ее элемент, начиная с первого символа, в szTarget. Цикл заканчивается, когда sourceIndex указывает на нулевой символ строки szSource. Функция concatstring () присоединяет завершающий нулевой символ к результирующей строке, и она выводится на экран.



Не забывайте добавлять завершающий нулевой символ к строке, которую вы создаете программно. Если при отображении строки на экране в ее конце появились непонятные "кракозябры" или если программа почему-то зависла, проверьте, не забыли ли вы добавить в строку этот волшебный нулевой символ.

Позаботьтесь о том, чтобы в первом символьном массиве хватило места для размещения результирующей объединенной строки. Следующие инструкции С++ выглядят весьма заманчиво:

```
char dash[] = "- ",-
concatString(dash, szMyName);
```

Однако корректно этот код работать не будет, потому что в массиве dash можно сохранить всего четыре символа. Поэтому функция concatString(> несомненно перейдет границы массива dash.

Функции С++ для работы со строками

Для работы со строками в С++ можно использовать стандартные библиотечные функции. Некоторые из них намного сложнее, чем может показаться с первого взгляда.

Вы можете написать собственные версии этих функций — например, в образовательных целях, как это было в случае с функцией объединения строк. Однако, используя функции из табл. 7.1, вы оградите себя от многих неприятностей и головной боли.

Таблица 7.1. Функции, обрабатывающие строки

НАЗВАНИЕ	ДЕЙСТВИЕ
int strlen(string)	Возвращает количество символов в строке (без учета нулевого символа)
char*" strcat (target, source)	Присоединяет строку source к концу строки target
char* strcpy(target, source)	Копирует строку source в target
char* strstr(source1, source2)	Находит первое вхождение строки source2 В source1
int strcmp(source1, source2)	Сравнивает две строки
int stricmp(source1, source2)	Сравнивает две строки без учета регистра символов

Чтобы использовать функции работы со строками, нужно добавить в начале программы директиву #include<string.h>.

В программе Concatenate вызов функции concatString () можно было бы заменить вызовом стандартной функции С++ strcat (). Это избавило бы от необходимости написания собственного кода.



Для некоторых эти функции могут показаться неинтересными и простыми. Подумаешь, строка добавляется к концу другой! Собственную функцию concatString() мы написали для того, чтобы на ее примере показать, как работают со строками стандартные функции С++.

Обработка символов типа wchar_t

В C++ на каждую переменную стандартного типа char выделяется 8-битовое поле, в котором можно представить 256 значений (от 0 до 255). Это 10 цифр, 26 строчных и 26 прописных букв. При этом остается более чем достаточно места для включения символов кириллицы.

Проблемы с этим типом возникают, когда необходимо включить в текст восточные алфавиты, особенно китайские и японские иероглифы. А в них символов, без преувеличения, тысячи, т.е. намного больше, чем можно представить в обычном 8-битовом множестве символов.

В C++ включена поддержка новейшего символьного типа wchar_t, или расширенного char. Хотя он не настолько встроен в язык, как тип char, многие функции C++ позволяют работать с ним. Например, wstrstr() умеет сравнить два символьных множества типа wchar_t. Если вы будете разрабатывать интернациональные приложения и захотите использовать восточные языки, вам понадобятся функции, которые работают с этим расширенным символьным типом. Но, так как этот вопрос намного сложнее, его подробное обсуждение выходит за рамки данной книги.

Устранение устаревших функций вывода

C++ предоставляет набор низкоуровневых функций ввода и вывода. Наиболее часто используется функция printf, которая осуществляет вывод форматированной строки на устройство стандартного вывода. В простейшем варианте функции передается единственный аргумент — выводимая строка:

```
printf("Строка для вывода на дисплей");
```

Функция printf может осуществлять управляемый вывод с помощью внедряемых в строку команд управления форматом, начинающихся со знака %. Например, следующий фрагмент выводит строку со значениями целой и действительной переменных:

```
int nInt = 1;
double doubleVar = 3.5;
printf("Целое значение равно %i; "
       "действительное значение равно %f",
       nInt, doubleVar);
```

Целое значение будет вставлено на месте %i, а действительное — на месте %f, и выводимая строка будет выглядеть следующим образом:

```
Целое значение равно 1; действительное значение равно 3.5
```

Тем не менее в книге используется более простой и менее подверженный ошибкам со стороны программиста способ вывода с помощью потоков.

Глава 8

Первое знакомство с указателями в C++

В этой главе...

- ✓ Что такое адрес
- ✓ Использование указателей
- ✓ Передача указателей функциям
- ✓ Использование кучи

По сравнению с другими языками C++ достаточно консервативен, он обладает собственным уникальным синтаксисом, благодаря которому программы, работающие с большим количеством переменных, можно реализовать гораздо компактнее с помощью указателей. Указатель — это переменная, которая содержит адрес другой переменной (т.е. ее расположение в памяти).

В этой главе представлены основы работы с указателями. Сначала рассматриваются концепции, с которыми необходимо обязательно ознакомиться для работы с указателями, затем поясняется синтаксис указателей и некоторые причины их высокой популярности в C++.

Что такое адрес

Очевидно, что каждая переменная C++ расположена где-то в памяти компьютера. Память разбита на байты, каждый из которых имеет свой адрес — 0, 1, 2 и т.д.

Переменная intRandy может находиться ПО адресу 0x100, a floatReader — по адресу 0x180 (адреса в памяти принято записывать в шестнадцатеричном виде).

Подобно людям, всякая переменная занимает некоторое место, более того по сравнению с другой переменной ей может понадобиться больше или меньше места (не будем вдаваться в подробности о том, кто и сколько места занимает). Количество памяти, выделяемое для переменных разных типов, приведено в табл. 8.1 (размер переменных приведен для процессора Pentium и компиляторов GNU C++ и Visual C++).

Таблица 8.1. Названия типов и их размеры в памяти

Тип переменной	Размер в памяти (в байтах)
int	4
long	4
float	4
double	8

В соответствии с этой таблицей написана тестовая программа, наглядно демонстрирующая расположение переменных в памяти (не обращайте внимания на неизвестный оператор & — будем пока просто считать, что он возвращает адрес переменной в памяти).

```
// Layout — эта программа призвана дать
//          читателю представление о
```

```
//          расположении переменных в памяти
#include <stdio.h>
#include <iostream.h>

int main(int intArgc, char* pszArgs[])
{
    int    m1;
    int    n;
    long   1;
    float  f;
    double d;
    int    m2;

    // ЕЫВОД в шестнадцатеричном виде
    cout.setf(ios::hex);

    // выводить адреса переменных
    // по очереди, чтобы показать размер
    // каждой переменной
    cout << "&m1 = Ox" << (long)&m1 << "\n";
    cout << "&n  = Ox" << (long)&n  << "\n";
    cout << "&l  = Ox" << (long)&l  << "\n";
    cout << "&f  = Ox" << (long)&f  << "\n";
    cout << "&d  = Ox" << (long)&d  << "\n";
    cout << "&m2 = Ox" << (long)&m2 << "\n";

    return 0;
}
```



Не волнуйтесь, если программа будет возвращать разные значения при каждом запуске. Просто каждый раз программа хранит переменные по разным адресам. Важна только связь между адресами переменных.

Сравнив расположение переменных, можно заключить, что п занимает четыре байта, 1 также занимает четыре байта и т.д. (в соответствии с приведенной таблицей).



И GNU C++, и Visual C++ выделяют одинаковое количество памяти под переменные одного типа.

Использование указателей

Переменная-указатель содержит адрес, обычно это адрес другой переменной. В табл. 8.2 приведены основные операторы для работы с указателями.

Таблица 8.2. Операторы для работы с указателями

ОПЕРАТОР	НАЗНАЧЕНИЕ
& (унарный)	Получить адрес переменной
* (унарный)	Операция разыменования — возвращает переменную, на которую указывает указатель (в выражении).
	Указатель на данный тип (в объявлении)

Пример работы с операторами приведен в следующем листинге:

```
void fn()
{
    int intVar;
    int* pintVar;

    pintVar = &intVar;  // теперь pintVar указывает на intVar
    *pintVar = 10;      // сохраняет 10 в переменой типа int
}                       // по адресу, находящемуся в pintVar
```

Функция fn () начинается с объявления переменной intVar; в следующей строке объявляется pintVar — указатель на переменную типа int.

Указатели объявляются как обычные переменные, но в объявление добавляется унарный оператор *, который может быть использован совместно с именем любого типа. В данной строке этот символ используется вместе с именем фундаментального типа int. Однако этот оператор может использоваться для добавления к любому имени переменной типа.

При написании программ желательно придерживаться соглашений об именах, в соответствии с которыми первый символ в названии переменной указывает на ее тип. Например, можно использовать n для int, d для double и т.д. С учетом этого соглашения имена указателей далее в книге будут начинаться с буквы p.

Унарный оператор & в выражении означает "взять адрес переменной". Таким образом, в первой строке приведенного кода находится команда сохранения адреса переменной intVar в переменной pintVar.

Представим себе, что функция fn () начинается с адреса 0x100, переменная intVar расположена по адресу 0x102, а указатель pintVar — 0x106 (такое расположение намного проще результатов работы программы Layout; на самом деле вряд ли переменные будут храниться в памяти именно в таком порядке).

Первая команда программы сохраняет значение &intVar (0x102) в указателе pintVar. Вторая строка отвечает за присвоение значения 10 переменной, хранящейся но адресу, который содержится в указателе pintVar (в нем находится число 0x102, т.е. адрес переменной intVar).

Сравнение указателей и почтовых адресов

Указатели похожи на адреса домов. Ваш дом имеет уникальный адрес, и каждый байт в памяти компьютера тоже имеет уникальный адрес. Почтовый адрес содержит набор цифр и букв. Например, он может выглядеть так: 123 Main Street (конечно же, это не мой адрес! Я не люблю нашествий поклонников, если только они не женского пола). Адрес переменной в памяти содержит только цифры (например, 123456).

Можно хранить диван в доме по адресу 123 Main Street, и точно так же можно хранить число в памяти по адресу 0x123456. Можно взять лист бумаги и написать на нем адрес — 123 Main Street. Теперь диван хранится в доме, который находится по адресу, написанному на листке бумаги. Так работают сотрудники службы доставки: они доставляют диваны по адресу, который указан в бланке заказа, независимо от того, какой именно адрес записан в бланке (я ни в коем случае не смеюсь над работниками службы доставки — просто это самый удобный способ объяснить указатели).

Использовав синтаксис C++, это можно записать так:

```
House myHouse;
House* houseAddress;
houseAddress = &myHouse;
*houseAddress = couch;
```

Эта запись обозначает следующее: myHouse является домом, a houseAddress — адресом дома. Надо записать адрес дома myHouse в указатель houseAddress и доставить диван по адресу, который находится в указателе houseAddress.

Теперь используем вместо дома переменную типа int:

```
int myInt;
int* intAddress;
intAddress = &myInt;
*intAddress = 10;
```

Аналогично предыдущей записи, это поясняется так: myInt — переменная типа int. Следует сохранить адрес raylnt в указателе intAddress и записать 10 в переменную, которая находится по адресу, указанному в intAddress.

Использование разных типов указателей

Каждое выражение, как и переменная, имеет свой тип и значение. Тип выражения sintVar — указатель на переменную типа int, т.е. это выражение имеет тип int*. При сравнении его с объявлением указателя pintVar становится очевидно, что они одинаковы:

```
int* pintVar = SintVar;     // обе части этого присвоения
                            // имеют тип *int
```

Аналогично pintVar имеет тип int*, а *pintVar — тип int:

```
*pintVar =10                // обе части этого присвоения
                            // имеют тип int
```

Тип переменной, на которую указывает pintVar, — int. Это эквивалентно тому, что если houseAddress является адресом дома, то, как ни странно, houseAddress указывает дом.

Указатели на переменные других типов объявляются точно так же:

```
double doubleVar
double* pdoubleVar = &doubleVar
*pdoubleVar =10.0
```

В компьютере класса Pentium размер указателя равен четырем байтам, независимо от того, на переменную какого типа он указывает[14].

Очень важно следить за соответствием типов указателей. Представьте, что может произойти, если компилятор в точности выполнит такой набор команд:

```
int n1;
int* pintVar;
pintVar = &n1;
*pintVar = 100.0;
```

Последняя строка требует, чтобы по адресу, выделенному под переменную размером в четыре байта, было записано значение, имеющее размер восемь байтов. На самом деле ничего страшного не произойдет, поскольку в этом случае компилятор приведет 100.0 к типу int перед тем, как выполнить присвоение.

Привести переменную одного типа к другому явным образом можно так:

```
int iVar;
double dVar = 10.0;
iVar = (int)dVar;
```

Так же можно привести и указатель одного типа к другому:

```
int* piVar;
double dVar = 10.0;
double* pdVar;
piVar = (int*)pdVar;
```

14

Вообще говоря, размер указателя зависит не только от типа процессора, но и от операционной системы, используемого компилятора и так называемой модели памяти создаваемой программы. — Прим. ред.

Трудно предсказать, что может случиться, если сохранить переменные одного типа по адресам, выделенным под переменные другого типа. Сохранение переменных, имеющих большую длину, вероятно, приведет к уничтожению переменных, расположенных рядом. Такая ситуация наглядно продемонстрирована с помощью программы LayoutError:

```
// LayoutError — демонстрирует результат
//              неаккуратного обращения с указателями
#include <stdio.h>
#include <iostream.h>

int main(int intArgc, char* pszArgs[])
{
    int    upper = 0;
    int    n     = 0;
    int    lower = 0;

    // выводим значения объявленных переменных
    cout << "upper = " << upper << "\n";
    cout << "n     = " << n     << "\n";
    cout << "lower = " << lower << "\n";

    // сохраняем значение типа double
    // в памяти, выделенной для int
    cout << "\nСохранение double в int\n";
    double* pD = (double*) &n;
    *pD = 13.0;

    // показываем результаты
    cout << "upper = " << upper << "\n";
    cout << "n     = " << n     << "\n";
    cout << "lower = " << lower << "\n";

    return 0;
}
```

В первых трех строках функции main() происходит объявление трех переменных типа int. Допустим, что в памяти эти переменные находятся друг за другом.

Следующие три строки выводят значения этих переменных на экран. Не удивительно, что все три оказываются равными нулю. После этого происходит присвоение *pD = 13.0;, в результате которого число, имеющее тип double, записывается в переменную n, имеющую тип int. Затем все три переменные снова выводятся на экран.

После записи действительного числа в целочисленную переменную n переменная upper оказалась "забитой" каким-то мусором.

На языке домов и адресов эта программа будет выглядеть так:

```
house* houseAddress = &"123 Main Street";
hotel* hotelAddress;
hotelAddress = (hotel*)houseAddress;
*hotelAddress = TheRitz;
```

Указатель houseAddress инициализирован как указатель на мой дом. Переменная hotelAddress содержит адрес отеля. После этого вместо адреса моего дома записывается адрес отеля. Затем отель "Ритц" устанавливается по адресу моего дома. Однако, поскольку "Ритц" куда больше моего дома, не удивительно, что он уничтожит не только мой дом, но и дома моих соседей (хоть что-то приятное в результате ошибки!).

Типизация указателей предохраняет программиста от неприятностей, связанных с сохранением данных большего размера в меньшем объеме памяти. Присвоение

*pintVar = 100.С не вызывает никаких проблем, поскольку С++ известно, что pintVar указывает на целочисленную переменную и приводит 100.0 перед присвоением к тому же типу.

Передача указателей функциям

Одним из путей использования указателей является передача аргументов функции. Для того чтобы понять всю важность этого метода, необходимо разобраться, как происходит передача аргументов функциям.

Передача аргументов по значению

Вы мсгли заметить, что обычно нельзя изменить значение переменной, которая передавалась функции как аргумент. Рассмотрим следующий фрагмент кода:

```
void fn(intArg)
{
    int intArg = 10;
    //здесь значение intArg равно 10
}
void parent(void)
{
    int nl = O;
    fn(nl);
    // здесь nl равно 0
}
```

Функция parent () инициализирует переменную nl нулем. После этого значение nl передается в качестве аргумента функции fn(). В fn () переменной intArg присваивается значение 10. тем самым в fn() осуществляется попытка изменить аргумент функции. Поскольку в качестве аргумента выступает переменная nl, можно ожидать, что после возврата в parent() эта переменная должна иметь значение 10. Тем не мснее nl остается равной 0.

Дело в том, что С++ передает функции не переменную, а значение, которое в момент вызова функции находится в переменной. При вызове функции происходит вычисление значения передаваемого функции выражения, даже если это просто переменная.



Обычно, экономя слова, многие говорят что-то вроде "передаем переменную x функции fn()". На самом деле это означает, что функции передается значение выражения x.

Передача значений указателей

Указатель, как и любая другая переменная, может быть передан функции в качестве аргумента.

```
void fn(int* pintArg}
{
  *pintArg = 10;
}

void parent(void)
{
   int r. = 0;
```

```
    fn(&n);          //так передается адрес n
                     // теперь n равно 10
}
```

В этом случае вместо значения n функции fn() передается адрес этой переменной. Чем отличается передача значения переменной от передачи значения указателя на переменную, станет понятно, если рассмотреть присвоение, выполняющееся в функции fn().

Предположим, что n находится по адресу 0x102. В этом случае функции fn() передается аргумент, равный 0x102. Внутри fn() присвоение *pintArg = 10 выполняет запись целого значения 10 в переменную типа int, которая находится по адресу 0x102. Таким образом, ноль в переменной л заменяется на 10, поскольку в данном случае 0x102 и есть адрес переменной n.

Передача аргументов по ссылке

В C++ возможна сокращенная запись приведенного выше фрагмента, которая не требует от программиста непосредственной работы с указателями. В приведенном ниже примере переменная n перелается *по ссылке.* При передаче по ссылке вместо значения переменной функции передается ссылка на переменную (*ссылка*, по сути, является синонимом слова *адрес)*.

```
void fn(int& intArg)
{
    intArg = 10;
}

void parent(void)
i
    int n = 0;
    fn(n)
    // теперь значение n равно 10
}
```

В этом примере функция fis() получает не значение переменной n, а ссылку на нее и, в свою очередь, записывает 10 в переменную типа int, на которую ссылается intArg.

Использование кучи

Куна (heap) — это блок памяти изменяемого размера, который при необходимости может использоваться программой. Далее в этом разделе поясняется, зачем нужна куча и как ею пользоваться.

Очевидно, что если можно передать функции указатель, то можно и вернуть его как результат работы функции. Функция, которая должна вернуть некоторый адрес, объявляется следующим образом:

```
double* fn(void);
```

При работе с возвращаемыми указателями следует быть очень осторожным. Чтобы понимать, чем чревато неаккуратное использование указателей, следует познакомиться с концепцией области видимости переменных (т.е. с тем, где именно от переменных остается только видимость...).

Область видимости

Кроме значения и типа, переменные в C++ имеют еще одно свойство — область видимости, т.е. часть программы, в которой эта переменная определена.

Рассмотрим следующий фрагмент кода:

```
// эта переменная доступна для всех функций
// и существует на протяжении всего
// времени работы программы
// (глобальная область видимости)
int intGlobal;

// переменная intChild доступна только в функции child()
// и существует только во время выполнения
// функции child() или вызываемой ею
// (область видимости функции)
void child(void)
{
    int intChild;
}

// переменная intParent имеет область видимости функции
void parent (void)
{
    int intParent = 0;
    child();

    int intLater = 0;
    intParent = intLater;
}

int main(int nArgs, char* pArgs[])
{
    parent();
}
```

Программа начинает выполнять функцию main (). В первой же строке main () вызывает функцию parent (). В этой функции выполняется объявление переменной intParent, которая имеет область видимости, ограниченную функцией. Такая переменная называется локальной и доступна только в этой функции.

Во второй строке parent () вызывается функция child(). Эта функция также объявляет локальную переменную — intChild, областью видимости которой являются функция child(). При этом intParent функции child() недоступна (и область видимости intParent не распространяется на функцию child ()), но сама переменная продолжает существовать.

После окончания работы функции child() переменная intChild выходит из области видимости и уничтожается, т.е. она не только недоступна, но и не существует (память, которую занимала эта переменная, возвращена в пул свободной памяти для дальнейшего использования).

После возврата из функции child() продолжается выполнение подпрограммы parent (), и в следующей строке объявляется переменная intLater, которая имеет область видимости, ограниченную функцией parent(). В момент возврата в функцию main() переменные intLater и intParent выходят из области видимости и уничтожаются.

Кроме локальных переменных, программист может объявлять переменные вне всех функций. Такие переменные называются глобальными и доступны из любого места программы (их область видимости — вся программа).

Поскольку intGlobal в приведенном коде объявлена глобально, она доступна на протяжении работы всей программы и внутри любой из трех функций.

Проблемы области видимости

Приведенный ниже фрагмент программы будет скомпилирован, но не будет корректно работать.

```
double child(void)
{
    double dLocalVariable;
    return fidLocalVariable;
)
void parent(void)
{
    double* pdLocal;
    pdLocal = child();
    *pdLocal = 1.0;
)
```

Проблема в том, что переменная dLocalVariable объявлена внутри функции child(). Следовательно, в момент возврата адреса dLocalVariable из child() самой переменной уже не существует и адрес ссылается на память, которая вполне может быть занята для каких-то других целей.



Ошибки такого типа встречаются довольно часто, а способы их появления весьма разнообразны. К сожалению, такой тип ошибки пропускается компилятором и зачастую не вызывает аварийной остановки программы. Программа может отлично работать большую часть времени, пока память, которая в прошлом выделялась под dLocalVariable, не будет выделена другой переменной. Труднее всего найти ошибки, проявляющиеся спонтанно.

Использование блока памяти

Ошибки области видимости возникают потому, что C++ освобождает выделенную для локальных переменных память автоматически. Для решения этой проблемы необходим блок памяти, контролируемый непосредственно программистом. В этом блоке можно выделять память под переменные и удалять их независимо от того, что по этому поводу "думает" C++. Такой блок памяти называется *кучей (heap)*.

Память в куче можно выделить, используя оператор new; он пишется вместе с типом объекта, под который нужно выделить память. Приведенный ниже пример выделяет из кучи память для переменной типа double.

```
double* child(void)
{
    double* pdLocalVariable = new double;
    return pdLocalVariable;
}
```

Теперь, несмотря на то что переменная pdLocalVariable имеет область видимости в пределах функции child(), память, на которую указывает эта переменная, не будет освобождена после выполнения функции. Выделение и освобождение памяти в куче осуществляется только явно. Освобождение памяти в куче выполняется с помощью команды delete.

```
void parent(void)
{
    // функция child() возвращает
    // адрес переменной в куче
    double* pdMyDouble = child();
```

```
    // сохранение значения в созданной переменной
    *pdMyDouble = 1.1;

    // ...

    // возврат памяти куче
    delete pdMyDouble;
    pdMyDouble = 0;

    // ...
}
```

В этой программе указатель, возвращенный функцией `child()`, используется для записи значения типа double в память, выделенную в куче. После того как функция выполнила все необходимые действия с этой памятью, она освобождается, а указатель pdMyDouble устанавливается равным нулю. Обнуление указателя не обязательно, но крайне желательно. В этом случае, если программист ошибется и попытается опять записать что-либо по адресу, на который указывает pdMyDouble, произойдет аварийный останов программы.



Ошибку, в результате которой происходит аварийный останов программы, найти гораздо проще, чем проявляющуюся спонтанно.

Глава 9

Второе знакомство с указателями

В этой главе...

- ✓ Операции с указателями
- ✓ Объявление и использование массивов указателей

Язык С++ позволяет работать с указателями так, как если бы они были переменными простых типов. (Концепция указателей изложена в главе 8, "Первое знакомство с указателями в С++".) Однако операции над указателями требуют знания некоторых тонкостей; именно они и рассматриваются в этой главе.

Операции с указателями

Некоторые из операций, описанных в главе 3, "Выполнение математических операций", могут быть применены к указателям. В этом разделе рассматривается применение арифметических операций при работе с указателями и массивами (с массивами вы познакомились в главе 7, "Хранение последовательностей в массивах"). В табл. 9.1 приведены базовые операции над указателями.

Таблица 9.1. Три операции над указателями

ОПЕРАЦИЯ	РЕЗУЛЬТАТ	ДЕЙСТВИЕ
`pointer+offset`	Указатель	Вычисляет адрес элемента, расположенного через `offset` элементов после `pointer`
`pointer-offset`	Указатель	Операция, противоположная сложению
`pointer2-pointerl`	Смещение	Вычисляет количество элементов между `pointerl` и `pointer2`

В этой таблице `offset` имеет тип `int` (здесь не приведены операции, близкие к сложению и вычитанию, такие как ++ и +=, которые также могут применяться к указателям).

Модель памяти, построенная на примере домов (так эффективно использованная в предыдущей главе), поможет понять, как работают приведенные в таблице операции с указателями. Представьте себе квартал, в котором все дома пронумерованы по порядку. Дом, следующий за домом 123 Main Street, будет иметь адрес 124 Main Street (или 122 Main Street, если вы идете в противоположную сторону, поскольку вы левша или живете в Англии).

Очевидно, что в таком случае через четыре дома от моего будет находиться дом с адресом 127 Main Street. Адрес этого дома можно записать как

```
123 Main Street + 4 = 127 Main Street
```

И наоборот, если поинтересоваться, сколько домов находится между домом 123 и 127, ответом будет четыре:

```
127 Main Street - 123 Main Street = 4
```

Понятно, что любой дом отстоит сам от себя на расстояние ноль домов:

```
123 Main Street - 123 Main Street = 0
```

Повторное знакомство с массивами в свете указателей

Обратимся к странному и мистическому миру массивов. Еще раз воспользуемся в качестве примера домами моих соседей. Массив тоже очень похож на городской квартал. Каждый элемент массива выступает в качестве дома в этом квартале. Дома — элементы массива — отсчитываются по порядку от начала квартала. Дом на углу отстоит на 0 домов от угла, следующий дом отстоит на 1 дом от угла и т.д.

Теперь представим себе массив из 32-х однобайтовых значений, имеющий имя charArray. Если первый элемент массива находится по адресу 0x110, тогда массив будет продолжаться вплоть до адреса 0x12f. Таким образом, элемент массива charArray[0] находится по адресу 0x110, charArray[1] — по адресу 0x111, charArray[2] — по адресу 0x112 и т.д.

Можно создать указатель ptr на нулевой элемент массива. После выполнения строки

```
ptr = &charArray[0];
```

указатель ptr будет содержать адрес 0x110. Можно прибавить к этому адресу целочисленное смещение и перейти к необходимому элементу массива. Операции над массивами с использованием указателей приведены в табл. 9.2. Эта таблица демонстрирует, каким образом добавление смещения n вызывает переход к следующему элементу массива charArray.

Таблица 9.2. Добавление смещения

СМЕЩЕНИЕ	РЕЗУЛЬТАТ	СООТВЕТСТВУЕТ
+0	0x110	charArray[0]
+1	0x111	charArray[1]
+2	0x112	charArray[2]
...	...	...
+n	0x110+n	charArray[n]

Как видите, добавление смещения к указателю на массив равнозначно переходу к соответствующему значению.

Таким образом, если char* ptr = &charArray[0];, то *(ptr + n) соответствует элементу charArray[n].



Поскольку * имеет более высокий приоритет, чем сложение, операция *ptr + n привела бы к сложению n со *значением,* на которое указывает ptr. Чтобы выполнить сначала сложение и лишь затем переход к переменной по указателю, следует использовать скобки. Выражение *(ptr + n) возвращает элемент, который находится по адресу ptr плюс *n* элементов.

В действительности соответствие между двумя формами выражений настолько строго, что C++ рассматривает элемент массива array[n] как *(ptr + n), где ptr указывает на первый элемент array. C++ интерпретирует array[n] как *(&array[0]+n). Таким образом, если дано char charArray[20], то charArray определяется как &charArray[0].

Имя массива, записанное без индекса элемента, интерпретируется как адрес нулевого элемента массива (или просто адрес массива). Таким образом, можно упростить приведенную выше запись, поскольку array[n] C++ интерпретирует как *(array + n).

Использование операций над указателями для адресации внутри массива

Концепция соответствия между индексацией массива и арифметикой указателей весьма полезна (если бы это было не так, здесь бы не было данного подраздела).

Например, функция displayArray(), которая выводит содержимое целочисленного массива, может быть реализована следующим образом:

```
// displayArray — отображает элементы массива,
// имеющего длину nSize
void displayArray(int intArray[], int nSize)
{
    cout << "Значения элементов массива равны:\п";

    for(int n = 0; n < nSize; n++)
    {
        cout « n << ": " << intArray[n] « "\n";
    }
    cout << "\n";
}
```

Эта версия функции использует операции над массивами, которые знакомы нам по предыдущим главам. Если воспользоваться для написания этой функции указателями, программа приобретет такой вид:

```
// displayArray — отображает элементы массива,
// имеющего длину nSize
void displayArray(int intArray[], int nSize)
{
    cout << "Значения элементов массива равны:\n";

    int* pArray = intArray;
    for(int n = 0; n < nSize; n++, pArray++)
    {
        cout << n « ": " << *pArray « "\n";
    }
    cout << "\n";
}
```

Этот вариант функции displayArray начинается с создания указателя на первый элемент массива intArray.



Буква p в начале имени переменной означает, что эта переменная являет" ся указателем[15].

После этого функция считывает все элементы массива по порядку. При каждом выполнении оператора for происходит вывод текущего элемента из массива intArray. Этот элемент находится по адресу pArray, который увеличивается на единицу при каждом выполнении цикла.

Можно сказать, что функция почти не изменилась и выполняет такие же операции, как и предыдущая версия, однако использование указателей — более распространенная практика, чем работа с массивами. По ряду причин программисты избегают работать с массивами.

Чаще всего указатели используются для работы с символьными массивами.

---

[15] *Однако это не более чем соглашение, и вы не обязаны начинать имена всех указателей с буквы p.* — Прим. ред.

Использование указателей для работы со строками

Строку можно назвать массивом символов, в котором последний символ равен нулю (язык C++ использует ноль как символ конца строки). Такие нуль-завершенные массивы можно рассматривать как отдельный тип (точнее, квази-тип), о котором шла речь в главе 7, "Хранение последовательностей в массивах". В C++ для работы со строками используются указатели. В приведенных ниже примерах показано, каковы отличия в работе со строками в случае применения массивов и указателей.

Отличия в работе со строками с использованием указателей и массивов

С помощью указателей можно работать с символьными массивами так же, как и с массивами любого другого типа. То, что в конце любой строки находится символ конца строки, делает их особенно удобными для работы с помощью указателей.

В главе 7, "Хранение последовательностей в массивах", рассматривалась функция concatString(), которая объединяла два символьных массива. Прототип этой функции был объявлен так:

```
void concatString(char szTarget[], char szSource[]);
```



Прототип описывает, какие аргументы принимает функция и какой тип имеет возвращаемый функцией результат. Прототип имеет такой же вид, как и объявление функции, однако в нем отсутствует тело функции.

Для обнаружения нуля в конце массива szTarget функция concatString () просматривает содержимое всего массива с помощью цикла while:

```
void concatString(char szTarget[], char szSource[])
{
    // найти конец первой строки
    int intTargetIndex = 0;
    while(szTarget[intTargetIndex])
    {
        intTargetIndex++;
    }
//...
```

Чтобы использовать для решения этой задачи указатели, следует объявить прототип функции так:

```
void concatString(char* pszTarget, char* pszSource);
```

А саму функцию необходимо переписать следующим образом:

```
void concatString(char* pszTarget, char* pszSource)
{
    // найти конец первой строки
    while(*pszTarget)
    {
        pszTarget++;
    }
    //...
```

В предыдущей версии функции concatString цикл while повторялся до тех пор, пока элемент szTarget[intTargetIndex] не оказывался равным 0. В варианте с указателями считывание и сравнение с нулем элементов массива происходит с помощью указателя pszTarget.



Выражение ptr++ представляет собой сокращенную форму записи ptr = ptr + 1.

После выхода из цикла while указатель pszTarget указывает на конечный элемент массива, содержащий ноль. Теперь сказать, что pszTarget указывает на массив, нельзя, поскольку он больше не указывает на начало массива.

Завершение функции concatstring()

В этом примере приведена готовая программа для объединения двух строк в одну:

```
// ConcatenatePtr — соединяет две строки и добавляет
//                  между ними " - ", используя для этого
//                   действия с указателями вместо
//                   индексов массива
#include <stdio.h>
#include <iostream.h>

void concatString(char* pszTarget, char* pszSource);

int main(int nArg, char* pszArgs[])
{
    // считать первую строку...
    char szStringl [256];
    cout « "Введите строку #1: ";
    cin.getline(szString1, 128);

    // ...теперь вторую...
    char szString2 [128];
    cout << "Введите строку #2: ";
    cin.getline(szString2, 128);

    // ...присоединить " — " к концу первой...
    concatString(szString1, " — " ) ;

    // ...теперь добавить вторую строку...
    concatString(szStringl, szString2);

    // ...и показать результат
    cout << "\п" « szStringl « "\п";

    return 0;
}

// concatString — присоединяет *pszSource к окончанию
//                строки *pszTarget
void concatString(char* pszTarget, char* pszSource)
{
    // найти конец первой строки
    while(*pszTarget)
    {
        pszTarget++;
    }

    // присоединяет вторую строку к концу первой
    // (и копирует ноль из второй строки в конец
    // созданной строки, чтобы было известно, где
```

```
    // заканчивается полученная строка)
    while(*pszTarget++ = *pszSource++)
    {
    }
}
```

Функция main() новой программы ничем не отличается от своей предшественницы, однако concatstring() существенно отличается от варианта с использованием массивов.

Как можно заметить, теперь объявление прототипа функции concatstring в начале программы содержит аргументы типа char*. Кроме того, первый цикл while в функции concatstring{}) ищет символ конца строки с помощью указателя pszTarget.

Исключительно компактный цикл в конце подпрограммы concatstring() выполняет присоединение символьного массива pszSource к концу массива pszTarget. В этом выражении вся работа по соединению осуществляется самим оператором while(), который выполняет ряд операций.

1. Считывает символ, на который указывает pszSource.
2. Увеличивает значение указателя pszSource для работы со следующим символом.
3. Записывает считанный символ в позицию, на которую указывает pszTarget.
4. Увеличивает значение указателя pszTarget для работы со следующим символом.
5. Выполняет тело цикла, если считанный символ не ноль.

После выполнения пустого тела программы управление опять передается оператору while(). Этот цикл будет выполняться, пока скопированный в pszTarget символ не окажется символом конца строки.

Почему при работе со строками пользуются указателями

Иногда некоторая запутанность работы с указателями вызывает у читателя вполне резонный вопрос: почему стоит пользоваться указателями? Иными словами, что делает использование указателя char* предпочтительнее более простого для чтения варианта с массивами и индексами?



Вариант функции concatenate() с использованием указателей гораздо распространеннее в C++, чем вариант с массивами.

Ответ следует искать отчасти в человеческой природе, отчасти в истории развития C++. Компилятор языка С, прародителя C++, в те времена, когда язык появился на свет, был довольно примитивен. Тогда компиляторы не были столь сложными, как сейчас, и не могли так хорошо оптимизировать код. Код `while(*pszTarget++ = *pszSource++){}` может показаться читателю сложным, однако после компиляции с помощью даже самого древнего компилятора он будет состоять буквально из нескольких машинных инструкций.

Старые компьютеры были не очень быстрыми по современным меркам. Во времена С экономия нескольких машинных инструкций значила очень много, что и привело к превосходству С над другими языками того времени, такими как FORTRAN, который не поддерживал работу с указателями.

Именно тогда и зародилась традиция писать компактные и эффективные, правда, подчас несколько загадочные на вид программы на C++, и с тех пор никто не хочет возвращаться к индексам.



Не надейтесь, что, написав сложное и запутанное выражение на С++, вы сэкономите несколько машинных команд. В С++ нет прямой связи между количеством команд в исходном и конечном коде. Сравните два набора команд в приведенном ниже фрагменте.

```
// выражение
*pszArray++ = '\0';

//и выражение
*pszArray2 = '\0';
pszArray2 = pszArray2 + 1;

// после компиляции могут давать
// одинаковое количество машинных инструкций
```

Во времена примитивных компиляторов, которые не оптимизировали код, первый вариант вызвал бы использование меньшего количества машинных команд. Однако современным оптимизирующим компилятором в обоих случаях будет сгенерирован идентичный код.

Операции с указателями других типов

Нетрудно сообразить, что szTarget + n указывает на элемент szTarget[n], если szTarget является массивом однобайтовых значений. Если szTarget начинается по адресу 0x100, то шестой элемент массива будет находиться по адресу 0x105.

Однако положение элемента в массиве становится не столь очевидным, если массив состоит из элементов типа int, которые занимают по четыре байта каждый. Если первый элемент такого массива находится по адресу 0x100, то шестой будет находиться по адресу 0x114 {0x100 + (5*4) = 0x114).

Но, к счастью для нас, выражение вида array + n будет всегда указывать на элемент array[n], независимо от размера элемента, поскольку в таком выражении С++ самостоятельно учитывает длину элемента.

И вновь обратимся за аналогией к моему дому. Третий дом от 123 Main Street будет иметь адрес 126 Main Street, независимо от размеров стоящих на Main Streel домов.

Отличия между указателями и массивами

Есть и несколько отличий в использовании массива и указателя. Во-первых, объявление массива вызывает выделение памяти для всего массива, тогда как объявление указателя не вызывает выделения памяти для массива.

```
void arrayPointer()
{
    // выделение памяти для 128 символов
    char charArray[128];

    // выделение памяти для указателя, но
    // не для объекта, на который он указывает
    char* pArray;
}
```

В этом примере для charArray выделяется 128 байт, а для pArray — четыре, ровно столько, сколько необходимо для хранения указателя.

Приведенная ниже функция работать не будет.

```
void arrayVsPointer()
{
     // этот фрагмент будет работать нормально
     char charArray[128];
```

```
    charArray[10] = 'O';
    * (charArray + 10) = 'O';

    // этот фрагмент не будет работать так, как надо
    char* pArray;
    pArray[10] = '0';
    * {pArray + 10} = '0';
}
```

Выражения charArray[10] и *(charArray + Ю) с позиции компилятора эквивалентны и вполне законны. Те же выражения с использованием pArray являются бессмысленными. Несмотря на то что для C++ они являются законными, pArray не инициализирован как указатель на массив, а значит, память была выделена только для указателя. Таким образом, pArray[10] и * (pArray + 10) указывают на неизвестные и непредсказуемые значения.



Неправильно инициализированные указатели обычно вызывают ошибку *нарушения сегмента (segment violation)*. Эту ошибку вы иногда встречаете в повседневной работе со своими любимыми приложениями в своей любимой (а может, и не очень) операционной системе.

Вторым отличием между указателями и индексами массива является то, что charArray — константа, тогда как pArray — нет. Приведенный ниже цикл for, который должен инициализировать значения элементов массива, тоже не будет работать.

```
void arrayVsPointer()
{
    char charArray[10] ;
    for ;int i = 0; i < 10; i++)
    {
        "charArray = '\0';      //эта строка имеет смысл...
    }   charArray++;           // ... а эта нет
}
```

Выражение charArray++ имеет не больше смысла, чем 10++. Правильно следует написать так:

```
void arrayVsPointer()
{
    char charArray[10];
    char* pArray = charArray;
    for (int i = 0; i < 10; i++)
    {
        *pArray = '\0';        // этот вариант будет
        pArray++;              // работать так как надо
    }
}
```

Объявление и использование массивов указателей

Если есть указатели на массивы, можно предположить, что существуют и массивы указателей. Именно их мы сейчас и рассмотрим.

Поскольку массив может содержать данные любого типа, он может состоять и из указателей. Массив указателей объявляется так:

```
int* pInts[10];
```

Таким образом, элемент pInts[0] является указателем на переменную типа int. Следовательно, приведенный ниже код корректен:

```
void fn()
{
    int n1;
    int* pints[3];
    pInts[0] = &n1;
    *pInts[0] = 1;
}
```

Как и этот:

```
void fn()
{
    int nl,n2,n3;
    int* pInts[3] = {snl, &n2, &n3,};
    for (int i = 0;i < 3; i++)
    {
        *pInts[i] = 0;
    }
}
```

И даже этот:

```
void fn()
{
    int nl,n2,n3;
    int* pInts[3] = {(new int),
                     (new int),
                     (new int)},
    for (int i = 0;i < 3; i++)
    {
        *pInts[i] = 0;
    }
}
```

В последнем варианте память под переменные выделяется из так называемой кучи, т.е. доступной программе памяти компьютера.

Массивы указателей чаще всего используются для работы с массивами строк. Приведенные далее примеры показывают, почему это удобно.

Использование массивов строк

Поскольку C++ поддерживает массивы указателей, значит, он должен поддерживать и массивы указателей на массивы. C++ действительно поддерживает такие конструкции, при этом количество повторений в принципе не офаничено (можно создать массив указателей на массив указателей на массив указателей...). Отдельный интерес представляют массивы указателей на строки. (Не забывайте, что строка является массивом символов.)

Допустим, мне понадобилась функция, возвращающая название месяца по его номеру. Например, если этой функции передать число 1, она вернет название первого месяца — "Январь". Номер месяца будет считаться неправильным, если он окажется меньше 1 или больше 12.

Эту функцию можно написать следующим образом:

```
// int2month() — возвращает название месяца
char* int2month(int nMonth)
{
    char* pszReturnValue;

    switch(nMonth)
    {
```

```
        case 1: pszReturnValue = "Январь";
                break;
        case 2: pszReturnValue = "Февраль";
                break;
        case 3: pszReturnValue = "Март";
                break;
        // и так далее...
        default: pszReturnValue = "Неверный номер месяца"
    }
    return pszReturnValue;
}
```



Оператор switch {> действует так же, как совокупность операторов if.

Эту задачу можно решить более элегантно, использовав номер месяца как индекс в массиве указателей, представляющих названия месяцев. Тогда программа приобретет такой вид:

```
// int2month() — возвращает название месяца
char* int2month(int nMonth)
{
    // проверка правильности номера месяца
    if (nMonth < 1 | | nMonth > 12)
    {
        return "invalid";
    }

    // nMonth имеет корректное значение
    // вернем имя месяца
    char* pszMonths[] = ("Ошибка",
                         "Январь",
                         "Февраль",
                         "Март",
                         "Апрель",
                         "Май",
                         "Июнь",
                         "Июль",
                         "Август",
                         "Сентябрь",
                         "Октябрь",
                         "Ноябрь",
                         "Декабрь"};
    return pszMonths[nMonth];
}
```

Сначала в этой программе проверяется корректность аргумента nMonth, т.е. что его значение лежит в диапазоне между 1 и 12 включительно (в предыдущей программе проверка производилась, по сути, оператором default). Если значение nMonth правильное, оно используется как смещение внутри массива, содержащего названия месяцев.

Доступ к аргументам main ()

Второй аргумент функции main () — массив указателей на строки. Эти строки содержат аргументы, передаваемые программе при вызове. Допустим, я ввел следующее в командной строке MS DOS:

```
MyProgram file.txt /w
```

MS DOS запустит программу, которая находится в файле MyProgram.exe, и передаст ей как аргументы file.txt и /w. Аргументы, начинающиеся с косой черты (/) или дефиса (-), обрабатываются операционной системой, как и любые другие: они передаются программе, чтобы та разбиралась с ними сама. Аргументы, которые начинаются с <, >, >> или | ! (а иногда и некоторые другие), представляют особый интерес для операционных систем и программе не передаются.

Аргументы программы являются одновременно аргументами функции main (). Переменная pszArgs, передаваемая main О, содержит массив указателей на аргументы программы, а nArgs — их количество.

Ниже приведен пример считывания аргументов из командной строки.

```
// PrintArgs — выводит аргументы программы
//            в стандартный вывод операционной системы
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // печатает начальную строку
    cout << "Аргументами программы " << pszArgs [0] « " являются\n" ;

    // Выводит аргументы программы
    for  (int i = 1;  i < nArg;  i++)
    {
        cout << i << ":  " « pszArgs[i]  «  "\n";
    }

    // вот и все
    cout << "Вот и все \п";
    return 0;
}
```

Как всегда, функция main() получает два аргумента. Первый — переменная типа int, которую я назвал nArgs. Эта переменная содержит количество передаваемых программе аргументов. Вторая переменная содержит массив указателей типа char*; ее я назвал pszArgs. Каждый из этих указателей ссылается на один из аргументов программы.

Если запустить программу PrintArgs с аргументами

PrintArgs arg1 arg2 arg3 /w

из командной строки MS DOS, nArgs будет равняться 5 (по количеству аргументов). Первый аргумент — имя самой программы. Таким образом, pszArgs [0] будет указывать на имя запускаемого файла, а остальные четыре указателя — на оставшиеся четыре аргумента (в данном случае это “arg1”, “arg2”, ...). Поскольку MS DOS никак не выделяет символ /, последний аргумент будет представлять собой строку "/w".

Глава 10

Прочие функциональные особенности

В этой главе...

- Зачем разбивать программу на модули
- Пример большой программы
- Разделение программы FunctionDemo
- Использование директивы #include
- Использование стандартных библиотек C++

Некоторые программы достаточно малы, чтобы без проблем размешаться в одном исходном .cpp-файле. Для большинства же промышленных программ это ограничение было бы слишком строгим. В этой главе показано, как разбить программу на несколько исходных файлов, каждый из которых можно отдельно тестировать и компилировать.

Зачем разбивать программу на модули

Программист может разбить программу на несколько исходных файлов, которые обычно называют модулями. Каждый из этих модулей можно компилировать отдельно, а затем объединить их все в процессе построения конечной программы.

Процесс объединения отдельно скомпилированных модулей называется *связыванием* или *компоновкой (linking)*.

Разделение программ на несколько небольших, удобных для редактирования и управления частей имеет определенные преимущества. Во-первых, это уменьшает время компиляции. GNU C++ или Visual C++ "проглотят" приведенные в этой книге программы и выдадут выполнимые файлы за несколько секунд. Большие программы требуют больше времени на компоновку. Ваш покорный слуга работал над проектами, полная компиляция которых занимала целую ночь.

Пересобирать программу целиком каждый раз, когда была изменена одна функция, очень неудобно. Намного проще откомпилировать одну функцию, организованную в виде отдельного модуля (можно составить модуль и из нескольких функций).

Во-вторых, гораздо проще понимать, писать и отлаживать программу, состоящую из нескольких хорошо продуманных модулей, каждый из которых логически объединяет несколько связанных между собой функций. Крайне трудно разобраться в содержимом одного объемистого модуля, охватывающего все относящиеся к программе функции. Кроме того, по мере разрастания модуль становится все более запутанным и в результате его приходится неоднократно переписывать.

Последний (и, наверное, самый главный) аргумент заключается в том, что небольшие модули можно использовать и при написании других программ.

flfuuiefi большой программы

Чтобы продемонстрировать большую программу, можно использовать, например... нет, эту нельзя, поскольку тогда в книге не хватит места для моего юмора (хотя, может быть, стоило вставить в книгу одну большую программу и больше ничего не писать?). Пожалуй, `FunctionDemo` из главы 6, "Создание функций", вполне подойдет на роль большой программы.

Вы, вероятно, помните, что FunctionDemo.`cpp` содержит следующий код:

```
// FunctionDemo — демонстрирует использование функций
//                путем выделения внутреннего
//                цикла программы NestedDemo
//                в отдельную функцию

#include <stdio.h>
#include <iostream.h>

// sumSequence — суммирует последовательность введенных
//               с клавиатуры чисел, пока
//               не будет введено отрицательное число.
//               Возвращает сумму введенных чисел
int sumSequence(void)
{
    // Вечный цикл
    int nAccumulator = 0;
    for (;;)
    {
        // Ожидание следующего числа
        int nValue = 0;
        cout << "Введите следующее число: ";
        cin  » nValue;

        // Если число отрицательное...
        if (nValue < 0)
        {
            // ...тогда выполнить выход из цикла
            break;
        }

        // ...в противном случае добавить число
        // к аккумулятору

        nAccumulator = nAccumulator + nValue;
    }

    // вернуть сумму
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{
    cout << "Эта программа суммирует последовательности \п"
         << "чисел. Ввод последовательности можно прекратить,\п"
         << "введя отрицательное число.\n"
         << "Остановить программу можно,\п"
         << "введя два отрицательных числа подряд\n";
```

```
    // накопление последовательности чисел...
    int nAccumulatedValue;
    do
    {
        // сложить числа, введенные
        // с клавиатуры
        cout << "\nВведите следующую последовательность\n";
        nAccumulatedValue = sumSequence();

        // вывести полученный результат
        cout << "\nСумма равна "
             <<nAccumulatedValue
             << "\n";

    // ...пока сумма не равна 0
    } while (nAccumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

Как и во многих других программах этой книги, в FunctionDemo суммируется последовательность введенных с клавиатуры чисел. Однако, в отличие от остальных программ, для сложения введенных чисел main () вызывает функцию sumSequence ().

Разделение программы FunctionDemo

Модуль FunctionDemo.cpp можно логически разделить на две функции, которые выполняют разные действия. Сейчас функция main() приглашает пользователя ввести числа и выполняет цикл, в котором накапливаются и выводятся суммы последовательностей чисел. Функция sumSequence () суммирует последовательность чисел и возвращает полученный результат.

Таким образом, программу можно разделить на модули, один из которых будет содержать функцию, выполняющую сложение, а второй — использовать эту функцию для сложения последовательности чисел и вывода суммы.

Для демонстрации этого метода новая версия программы FunctionDemo разбита на два модуля: первый содержит функцию sumSequence (), а второй — функцию main ().

Конечно, приведенная программа очень мала. И хотя sumSequence () можно разделить еще на несколько модулей, разбивать FunctionDemo не имеет смысла ни для упрощения работы, ни для ускорения компиляции. Этот пример демонстрирует механизм разбиения программы на несколько частей и не более того.

Отделение модуля sumSequence ()

Функцию sumSequence() легко отделить от остальной программы. Приведенный ниже листинг SeparateModule.cpp содержит единственную функцию sumSequence():

```
// SeparateModule — демонстрирует, как можно разбить
//                  программу на несколько модулей,
//                  упростив написание и проверку;
//                  этот модуль содержит функцию,
```

```
//                    вызываемую из main()

#include <stdio.h>
#include <iostream.h>

// sumSequence - суммирует последовательность введенных
//              с клавиатуры чисел, пока
//              не будет введено отрицательное число.
//              Возвращает сумму введенных чисел
int sumSequence(ostream& out, istream& in)
{
    // Вечный цикл
    int nAccumulator = 0;
    for(;;)
    {
        // Ожидание следующего числа
        int nValue = 0;
        out << "Введите следующее число: ";
        in  >> nValue;

        // Если число отрицательное...
        if (nValue < 0)
        {
        // ...тогда выполнить выход из цикла
            break;
        }

        // ...в противном случае добавить число
        // к аккумулятору
        nAccumulator = nAccumulator + nValue;
    }

    // вернуть сумму
    returnnAccumulator;
}
```

Структура модуля SeparateModule.cpp почти такая же, как и других программ, однако эта программа отличается от остальных отсутствием функпии main(). Этот модуль будет скомпилирован, но на последнем этапе (компоновки) появится сообщение об ошибке.

Последний этап сборки программы называется связыванием, поскольку в этот момент разные модули связываются в единую конечную выполняемую программу.

Функция sumSequence () выглядит почти так же, как и в программе FunctionDemo из главы 6, "Создание функций", с одним небольшим отличием. В предыдущей версии ввод осуществлялся с помощью потока ввода cin, а вывод — с помощью потока cout. Для придания большей универсальности функции передаются имена потоков ввода и вывода.

Объект cin имеет тип istream, однако тот же тип могут иметь и файлы, отличные от стандартного ввода. Поскольку функции можно передать любой поток ввода и вывода, sumSequence () может быть использована для работы с любым типом потока, например с внешними файлами. В главе 26, "Использование потоков ввода-вывода", этот вопрос рассматривается подробнее.

Может показаться, что передача объектов ввода-вывода только вносит ненужную путаницу в программу, а путаницы следует избегать. Но в данном случае небольшая путаница — цена гибкости программы, обеспечивающая возможность дальнейшего использования этих объектов в других программах.

Создание модуля MainModule.cpp

Поскольку sumSequence () теперь находится в отдельном модуле, в MainModule . cpp остается только функция main ():

```
// MainModule- демонстрирует, как можно разбить
//             программу на несколько модулей,
//             упростив написание и проверку;
//             этот модуль содержит функцию main()
#include <stdio.h>
#include <iostream.h>

int sumSequence (ostream& out, istream& in);

int main(int nArg, char* pszArgs[])
{
cout << "Эта программа суммирует последовательности \n"
     << "чисел. Ввод последовательности можно прекратить,\n"
     << "введя отрицательное число.\n"
     << "Остановить программу можно,\n"
     << "введя два отрицательных числа подряд\n";

    // накопление последовательности чисел...
    int nAccumulatedValue;
    do
    {
        // сложить числа, введенные
        // с клавиатуры
        cout << "\nВведите следующую последовательность\n";
        nAccumulatedValue = sumSequence(cout, cin) ;

        // вывести полученный результат
        cout << "\nСумма равна "
             << nAccumulatedValue
             << "\n";

    // ...пока сумма не равна 0
    } while (nAccumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

В этом варианте программы отсутствует функция sumSequence (), но добавлена строка объявления прототипа:

```
int sumSequence(ostream& out, istream& in);
```

Использование прототипов функций пояснялось в главе 6, "Создание функций".

Поскольку тела функции в данном модуле программы нет, необходимо добавить ее прототип для того, чтобы можно было воспользоваться данной функцией, несмотря на то что сама она располагается в другом месте.

Создание файла проекта

Теперь можете открыть исходные файлы SeparateModule.cpp и MainModule. cpp в редакторе rhide. После этого выберите команду меню Compile⇨Make или нажмите <F9>. Редактор rhide скомпилирует оба файла и объединит их в программу с именем aout.exe (к счастью, ее всегда можно переименовать).

Создание файла проекта в GNU C++

Возможность скомпилировать все относящиеся к одной программе модули, открывая их, имеет один важный плюс: это очень просто. Однако, если в программе много модулей, необходимость открывать каждый из них в отдельности становится проблемой. Чтобы избежать этой работы, можно создать файл, в котором будут содержаться имена всех модулей, которые нужно скомпилировать и скомпоновать.

Для этого выполните ряд действий.

1. Закройте все открытые файлы и выберите Project⇨Open Project.
2. Введите имя проекта, например Separate (это имя не имеет особого значения — используйте любое понравившееся вам слово). В нижней части экрана появится окно проекта с одним элементом `<empty>`.
3. Выберите Project⇨Add Item. Появится окно, показывающее содержимое текущего каталога.
4. Выберите файл `MainModule.cpp`.
5. Щелкните на кнопке Cancel, чтобы закрыть текущее окно. Проект Separate создан.
6. Для компиляции выберите из меню Compile⇨Make.

Еще одно достоинство файла проекта в том, что он позволяет `rhide` сохранять в нем дополнительную информацию о программе, хотя и не такую богатую, как в случае использования Visual C++.

Создание файла проекта в Visual C++

Если при работе с этой книгой вы используете Visual C++, выполните следующее.

1. Выберите File⇨Close Workspace, чтобы закрыть все открытые файлы проектов.
2. Откройте исходный файл MainModule.cpp и щелкните на кнопке Compile (заметьте — *не* на кнопке Make).

   Если вы щелкнете на кнопке Make, ничего страшного не произойдет, просто на этапе связывания возникнет ошибка.
3. Visual C++ запросит подтвердить создание файла проекта. Подтверждение необходимо, поскольку Visual C++ не может работать с отдельным `.cpp`-файлом, который не входит ни в один проект.

   Теперь у нас есть проект, содержащий один файл — MainModule.cpp.
4. Если рабочее пространство пока не открыто, откройте его, выбрав из меню View⇨Workspace.

   При этом должно появиться окно, содержащее две вкладки: окно просмотра классов (Class View) и окно просмотра файла (File View). Эти окна представляют два пути просмотра проекта. В окне просмотра файла показано содержимое `.cpp`-файла.
5. Перейдите в окно просмотра файла в окне проекта, воспользовавшись соответствующей вкладкой.
6. Щелкните правой кнопкой мыши на файлах Mainmodule, при этом появится выпадающий список файлов, составляющих проект MainModule. Сейчас в проект входит только один файл — `MainModule.cpp`.

7. Выберите пункт меню Add Files, после чего появится окно открытия файлов, которое имеет такой же вид, как, например, в Microsoft Word.
8. Выберите файл SeparateModule.cpp для добавления его в проект. Теперь в списке файлов проекта должны находиться два файла — MainMcdule.cpp и SeparateModule.cpp.
9. Щелкните на кнопке Build для сборки выполнимого файла проекта.

использование директивы #include

Для того чтобы в функции main() можно было использовать функцию sumSequence(), необходимо включить ее прототип в начале программы. К сожалению, при вводе имени прототипа возможна опечатка или какая-либо другая ошибка. А если эта функция используется в нескольких модулях, программист должен объявить прототип в каждом из них, поэтому вероятность опечатки возрастает.

С++ предоставляет механизм, который помогает справиться с проблемой объявления прототипов. Программист может создать отдельный файл, который включается в программу на этапе компиляции. Подключаемые файлы работают следующим образом.

1. Создайте файл с именем SeparateModule.h, содержащий объявление прототипа sumSequence(). Расширение .h обозначает, что этот файл подключаемый:

```
// SeparateModule.h — включает объявление прототипа функции
//                    из модуля SeparateModule.cpp
int sumSequence(ostream& out, istream& in);
```

2. Отредактируйте файл MainModule.cpp, вставив директиву включения файла SeparateModule.h вместо объявления прототипа.

```
// MainModule- демонстрирует, как можно разбить
//             программу на несколько модулей,
//             упростив написание и проверку;
//             этот модуль содержит функцию main()

#include <stdio.h>
#include <iostream.h>

// объявление подключения внешних прототипов
#include "SeparateModule.h"

int main(int nArg, char* pszArgs[])
{
```

Директива ttinclude дает понять компилятору, что вместо нее здесь следует разместить содержимое файла SeparateModule.h. Таким образом, после подключения этого файла программа приобретет такой же вид, как и раньше.



Директива #include должна начинаться с первого символа строки.

Включение одного и того же файла в один модуль встречается гораздо чаще, чем вы предполагаете. Бывает, что в файл включается другой файл, в который включается третий, а в него четвертый, в который включается первый.

Это не представляет проблемы, если в .h-файл входят только прототипы функций и определения #define. Считается признаком дурного тона (и часто приводит к серьезным ошибкам) объявлять в подключаемых файлах глобальные переменные или создавать в них функции.

Повторных объявлений можно избежать, воспользовавшись директивой #ifdef. Она требует включения всех команд вплоть до #endif, если аргумент директивы #ifdef был определен ранее (#ifndef действует наоборот— если аргумент не был определен).

```
//MyInclude.h
//Проверяем, была ли использована директива #define MyModule_h
//до этого. Если нет, значит, этот подключаемый файл во время
//компхляпии вызывается впервые
#ifndef MyModule_h

//Определим MyModule_h с тем, чтобы при следующем
//подключении этого файла из той же программы
// реального включения файла не было
#define MyModule_h

//Сюда можно вставлять все, что вы хотите разместить
//во включаемом файле

//закрытие #ifndef в конце файла
#endif
```



Эта проверка будет проводиться во время компиляции, но не во время выполнения программы.

Использование стандартных библиотек C++

Теперь вам должно быть понятно, почему во всех примерах, приведенных в этой книге, используются директивы #include <stdio.h> и #include <iostream.h>. Эти подключаемые файлы содержат объявления всех используемых в программах стандартных функций.

Обратите внимание, что имена стандартных .h-файлов заключены в угловые скобки, тогда как при объявлении локальных .h-файлов используются обычные кавычки. Единственное отличие между этими типами объявлений состоит в том, что C++ начинает искать файлы, заключенные в обычные кавычки, в текущем каталоге (каталоге проекта), а поиск файлов, заключенных в угловые скобки, происходит в предопределенных каталогах, содержащих подключаемые файлы. С помощью настроек проекта программист может определить, где именно должен проводиться поиск включаемых файлов.

Глава 11

Отладка программ на C++

В этой главе...

- ✓ Определение типа ошибки
- ✓ Использование отладочной печати
- ✓ Использование отладчика
- ✓ Первая программа BUDGET

Не часто случается (особенно с "чайниками"), что программа идеально работает с первого раза. Крайне редко удается написать нетривиальную программу и не допустить ни одной ошибки.

Чтобы избавиться от ошибок, можно пойти двумя путями. Первый — стирание программы и написание ее заново, а второй — поиск и исправление ошибки. Освоение первого пути я оставляю читателю, а в этой главе расскажу о том, как выследить и исправить ошибку в программе.

Определение tfauta ошибки

Можно выделить два типа ошибок: те, которые компилятор может найти, и те, которые не может, Первый тип называют *ошибками компиляции* (compile-time error). Их довольно легко найти, поскольку компилятор сам указывает место в программе, где встретилась ошибка. Правда, иногда описание ошибки бывает не совсем точным (компьютер так легко сбить с толку!), однако, зная капризы своего компилятора, нетрудно разобраться в его жалобах.

Ошибки, которые компилятор не может найти, проявляются при запуске программы и называются *ошибками времени исполнения* (run-time error). Их найти намного труднее, поскольку, кроме сообщения об ошибке, нет и намека на то, какая именно ошибка возникла и где (сообщения, которые генерируются при возникновении ошибок выполнения, вполне достойны "звания" ошибочных).

Для выявления "жучков" в программе обычно используется два метода. Первый — добавить отладочные команды, выводящие ключевые значения в ключевых точках программы. Увидев значения переменных в месте возникновения ошибки, можно понять, что именно неправильно в данной программе. Второй метод заключается в использовании специальной программы — отладчика. Отладчик позволяет отслеживать процесс выполнения программы.

использование отладочной печати

Добавление команд вывода в ключевых точках помогает понять, что происходит в программе, и называется методом отладочной печати (иногда именуемым WRITE). Метод WRITE появился во времена, когда программы писались на языке FORTRAN, в котором вывод осуществляется с помощью команды WRITE.

Приведенная ниже "дефектная" программа наглядно демонстрирует применение отладочных команд. Эта программа должна считывать последовательность чисел с клавиатуры и выводить их среднее арифметическое значение. Однако она не делает этого, поскольку содержит две ошибки, одна из которых вызывает аварийный останов, а вторая приводит к неправильному результату.

```
// ErrorProgram — эта программа усредняла бы
//                ряд чисел, если бы не содержала
//                одну невыполнимую ошибку
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << "Эта программа содержит ошибки!\n";

    // аккумулирует ряд чисел, пока
    // пользователь не введет отрицательное число,
    // после чего выводит среднее
    int nSum;
    for (int nNums = 0; ; )
    i
        // ждет ввода следующего числа
        int nValue;
        cout << "\nВведите следующее число:";
        cin  >> nValue;

        // если введенное число меньше нуля...
        if {nValue < 0)
        (
            // ...то вывести среднее
            cout << "\пСреднее равно: "
                 << nSum/nNums
                 << "\п";
            break;

        }

        // если число больше нуля,
        // сложить его с аккумулятором
        nSum += nValue;
    }
    return 0;
}
```

После ввода этой программы создайте выполнимый файл (клавиша <F9>).

Запустите эту программу и введите числа 1, 2 и 3, а затем -1. Вы ожидаете увидеть, что их среднее равно двум? Вместо этого очевидного результата будет выдано довольно непривлекательное сообщение об ошибке, показанное на рис. 11.1.

Выявление "жучка" № 1

Сообщение об ошибке, приведенное на рис. 11.1, выглядит весьма внушительно. На самом деле большая часть информации в этом сообщении бесполезна для нас (как и для многих других). Но во второй строке есть намек на источник ошибки: Division by zero at ... (Деление на ноль в ...). Как можно понять, во время выполнения программы произошла попытка деления какой-то переменной на ноль (крайне информативно, не правда ли?). Определить источник ошибки не так уж

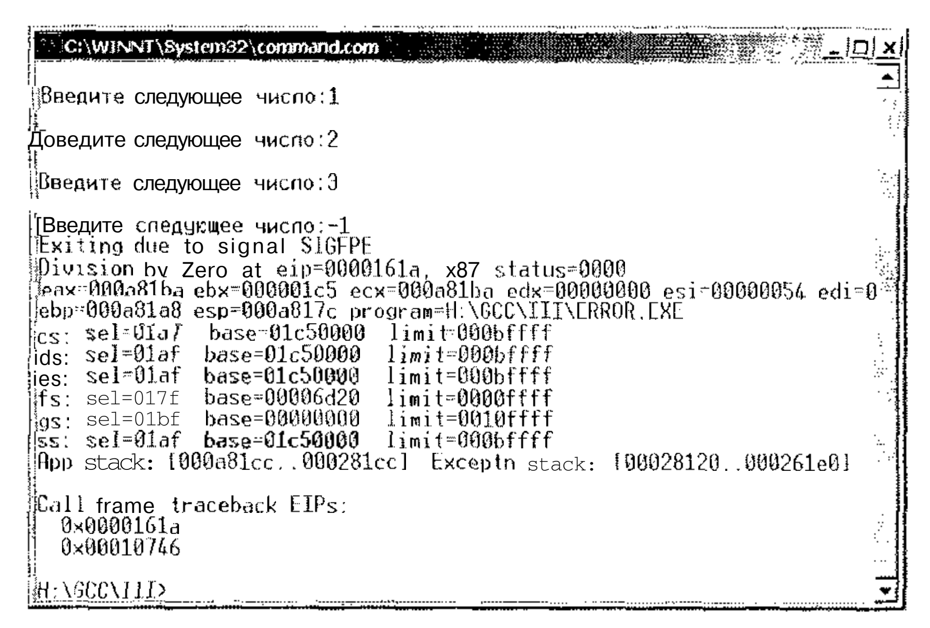


Рис. 11.1. Выполнение первоначальной версии программы прерывается в результате ошибки

и просто. Например, процессор мог "потерять" выполняемый код и продолжить выполнение машинных инструкций, не относящихся к программе (такое тоже иногда случается). Процессор мог использовать инструкцию деления как повод для генерации сообщения об ошибке, скрыв таким образом настоящий ее источник (программы с ошибками похожи на поезд, который сошел с рельсов: программа не остановится, пока не произойдет что-то действительно важное).

О таких программах иногда говорят, что они "стартовали в космос". Ну, а поскольку к ним прикрепили отдельный ярлык, значит, они встречаются довольно часто.

Следующим шагом будет запуск программы из среды разработчика, поскольку иногда среды разработчика, например Visual C++ или GNU C++, могут помочь в поиске ошибки.

Далее приведен пример с использованием GNU C++ (работа в Visual C++ очень похожа на описанную ниже).

Откройте программу в редакторе `rhide`, соберите и запустите ее, нажав клавиши <Ctrl+F9>. Введите 1, 2 и 3, а затем -1, и программа снова аварийно завершится.



Для устранения ошибки сначала необходимо найти, какой именно фрагмент кода ее вызывает. Зная расположение ошибки, вы сможете воспроизвести ее во время отладки, а Также будете знать, когда ошибка уже исправлена.

После запуска программы и возникновения ошибки rhide выдает окно, содержащее сообщение "Program exiting code 255(0xff)" ("Программа завершилась с кодом выхода 255(0xff)"), как показано на рис. 11.2. Я многого не знаю, но мне точно известно, что код нормального завершения нашей программы — 0. Невооруженным глазом видно, что этот код завершения не равен нулю, а значит, что-то пошло не так. Хотя я и не знаю, что именно означает код выхода 0xff.

Щелкните на кнопке OK, и rhide кроме окна редактирования программы откроет еще два окна.

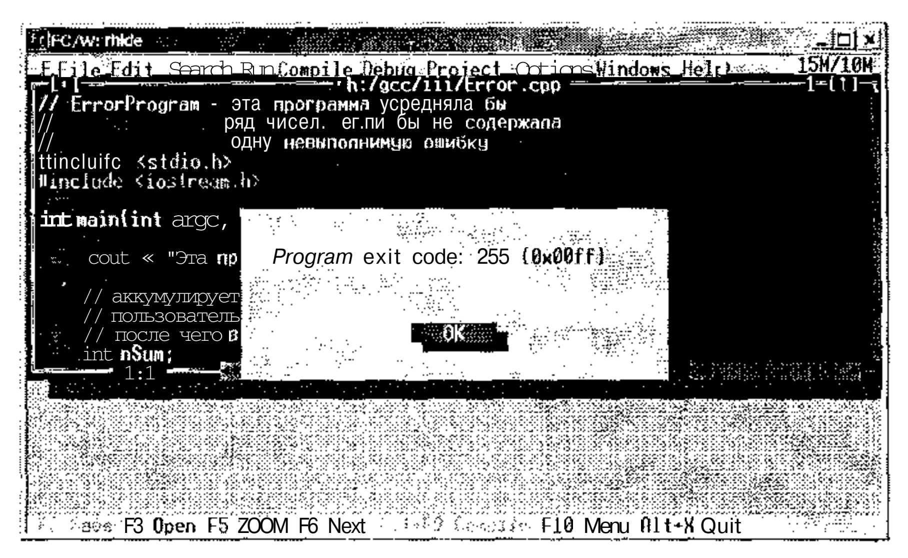


Рис. 11.2. Код выхода Oxff означает, что программа завершилась аварийно



Вы можете не видеть всех трех окон одновременно, поскольку одно из них может быть скрыто другими. Чтобы переключиться в нужное окно, используйте клавишу <F6>.

В третьем окне находится сообщение об ошибке, которое было создано во время запуска программы, но мы обратимся ко второму окну, показанному на рис. 11.3.

```
FC/W: rhide
File Edit Search Run Compile Debug Project Options Windows Help   15M/11M
h:/gcc/iii/Error.cpp
// ErrorProgram - эта программа усредняла бы
//                ряд чисел, если бы не содержала
//                одну невыполнимую ошибку
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout « "Эта программа содержит ошибки!\n";

    // аккумулирует ряд чисел, пока
    // пользователь не введет отрицательное число,
    // после чего выводит среднее
    int nSum;

Call frame traceback:
Error.cpp(28) in function main
in function crt1_startup+178

Enter JUMP to source F5 Zoom F6 Next F10 Menu Alt+X Quit
```

Рис. 11.3. Редактор rhide способен вычислить, в какой части программы возникла ошибка

Сообщение "Call frame traceback" ("Отслеживание кадров вызовов") звучит так же, как сообщение о шпионском радиоперехвате. Система выполняет просмотр адресов всех функций, вызванных программой, начиная с самой первой. В данном случае

ошибка находится в main {}, которая была вызвана функцией с названием __crtl_startup (крайне содержательно, не правда ли?). Итак, ошибка возникла в строке 28 исходного файла ErrorProgram.cpp. Это уже теплее...

Обратимся к строке 28 исходного файла:

```
cout << "\nAverage is : "  // строка 26
     << nSum/nNums          // строка 27
     << "\n";               // строка 28
```

Думаю, никто из читателей не видит в строке 28 операции деления. Я тоже не вижу. Дело в том, что при компиляции С++ собирает все выражения до точки с запятой в одну строку, т.е. строки 26, 27 и 28 являются частями одной строки. Таким образом, все команды в строках 26—28 будут рассматриваться компилятором как одна 28-я строка.

Теперь вы знаете, что ошибка возникла в результате деления в строке 27. Таким образом, можно заключить, что в момент деления переменная nNums была равна нулю. Эта переменная должна содержать количество введенных чисел. Просмотрев программу, можно увидеть, что nNums была инициализирована нулем, но после этого ее значение не увеличивалось. Переменная nNums должна была увеличиваться во время выполнения оператора for, так что для правильной работы программы нужно изменить строку, содержащую оператор for, следующим образом:

```
for (int nNums = 0; ;nNums++)
```

Выявление "жучка" № 2

Теперь, когда найдена и исправлена ошибка № I, можно запустить программу, введя числа, заставившие ее в прошлый раз аварийно завершиться. На этот раз сообщение об ошибке не появится и программа вернет нулевой код выхода, но будет работать не так, как ожидалось. Вместо так горячо ожидаемой двойки будет выведено какое-то нелепое число.

```
Эта программа содержит ошибки!

Введите следующее число: 1
Введите следующее число: 2
Введите следующее число: 3
Введите следующее число: -1
Среднее равно: 22 952 3
```

Как С++ находит ошибку в исходном коде

Сообщение об ошибке, полученное во время запуска программы из MS DOS или Windows, было не очень информативным (особенно по сравнению с Visual С++ или rhide, которые смогли указать, где именно возникла ошибка). Как же среды разработки находят источник ошибки?

Компилятор С++ поддерживает два режима построения программ. По умолчанию С++ строит программу в так называемом отладочном режиме. При этом компилятор добавляет в машинный код информацию о том, к какой строке исходного кода относится та или иная машинная инструкция, и тогда при запуске программы можно узнать, что, например, строка 200 машинного кода отвечает строке 16 исходной программы.

При возникновении ошибки деления на ноль С++ по номеру строки машинного кода отслеживает, в каком месте исходной программы находилась ошибка.

Такая отладочная информация занимает довольно много места. Поэтому при подготовке окончательной версии программы следует указать компилятору, что генерация исполняемого кода должна проводиться без отладочной информации.

Очевидно, какая-то из переменных — nNums или nSum (а возможно, и обе) содержит неверное значение. Для того чтобы исправить ошибку, необходимо узнать, какая именно из этих переменных содержит неверную информацию. Не помешало бы также знать, что содержится в переменной nValue, поскольку она используется для подсчета суммы в nSum.

Для этого воспользуемся методом отладочной печати. Чтобы узнать значения nValue. nSum и nNums, перепишите тело цикла for так, как показано в следующем листинге:

```
for (int nNums = 0; ;nNums++)
{
        // ждет ввода следующего числа
        int nValue;
        cout << "\nВведите следующее число:";
        cin  >> nValue;

        // если введенное число меньше нуля...
        if (nValue < 0)
        {
            // ...тогда вывести среднее
            cout << "\nСреднее равно: "
                 << nSum/nNums
                 << "\n";
            break;

        }
        // вывести отладочную информацию
        cout << "nSura = " << nSum  << "\n";
        cout << "nNums= " << nNums  << "\n";
        cout << "nValue= "<< nValue << "\n";
        cout << "\n";

        // если число больше нуля,
        // сложить его с аккумулятором
        nSum += nValue;
}
```

Обратите внимание на то, что информация о состоянии отслеживаемых переменных nValue, nSum и nNums выводится в каждом цикле.

Ответ программы на ввод уже привычных 1, 2, 3 и -1 приведен ниже. При первом же проходе nSum принимает какое-то несуразное значение, хотя оно должно равняться нулю (поскольку к этой переменной пока что ничего не прибавлялось).

```
Эта программа содержит ошибки!

Введите следующее число:1
nSum = -858993460
nNums= 0
nValue= 1

Введите следующее число:2
nSum = -858993459
nNums= 1
nValue= 2

Введите следующее число:3
nSum = -858993457
nNums= 2
nValue= 3
```

```
Введите следующее число:
```

Внимательно присмотревшись к программе, можно заметить, что nSura была объявлена, но не *проинициализирована*. Для того чтобы исправить эту ошибку, объявление переменной необходимо изменить следующим образом:

```
int nSum = 0;
```

Примечание. Пока переменная не проинициализирована, ее значение непредсказуемо.

Теперь, когда вы нашли все ошибки, перепишите программу так, как показано в следующем листинге:

```
// ErrorProgram — эта программа усредняет
//                ряд чисел и не содержит
//                ошибок
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << "Эта программа работает!\n";

    // аккумулирует ряд чисел, пока
    // пользователь не введет отрицательное число,
    // после чего выводит среднее
    int nSum = 0;

    for (int nNums = 0; ;nNums++)
    {
        // ждет ввода следующего числа
        int nValue;
        cout << "\nВведите следующее число:";
        cin  >> nValue;

        // если введенное число меньше нуля...
        if (nValue < 0)
        {
            // ...тогда вывести среднее
            cout << "\nСреднее равно:"
                 << nSum/nNums
                 << "\п";
            break;

        }

        // если число больше нуля,
        // сложить его с аккумулятором
        nSum += nValue;
    }

    return 0;
}
```

Теперь вывод программы будет правильным.

```
Эта программа работает!

Введите следующее число:1
Введите следующее число:2
```

```
Введите следующее число:3
Введите следующее число:-1

Среднее равно: 2
```

Протестировав эту программу другими наборами чисел, я убедился, что она работает без ошибок.

Использование отладчика

В небольших программах метол отладочной печати работает довольно неплохо. Добавление отладочных команд — достаточно простой и не влияющий на время компиляции способ нахождения ошибок, с помощью которого можно быстро отыскать ошибку, если программа невелика.

В действительно больших программах зачастую программист даже не знает, куда нужно добавлять отладочные команды. Работа по добавлению отладочных команд, перезапуску программы, повторному добавлению отладочных команд и так далее становится утомительной. Кроме того, после каждого переписывания программу нужно собирать заново. Не забывайте, что в большой программе один только процесс сборки может занять немало времени.

В конце концов, с помощью этого метода почти невозможно найти ошибку, связанную с указателями. Указатель, выведенный на экран в шестнадцатеричном виде, мало о чем скажет вам, и, пока программист поймет, что нужно сделать для исправления ошибки, программа успеет морально устареть.

Второй, более изощренный метод — использование отдельной утилиты, которая называется отладчиком. С помощью отладчика можно избежать трудностей, возникающих при использовании методики отладочной печати (однако, если вы хотите использовать отладчик, вам придется научиться с ним работать).

Что такое отладчик

Отладчик — это утилита, встроенная, например, в rhide или Microsoft Visual C++ (в этих приложениях разные программы отладчиков, однако работают они по одному принципу).

Программист управляет отладчиком с помощью команд так же, как, например, при редактировании или компиляции программы. Команды отладчика можно выполнять с помощью контекстных меню или горячих клавиш.

Отладчик позволяет программисту контролировать работу программы по ходу ее выполнения. С помощью отладчика можно выполнять программу в пошаговом режиме, останавливать ее в любой точке и просматривать содержимое любой переменной.

Чтобы оценить удобство отладчика, его нужно увидеть в действии. В этом разделе преимущества отладчика показаны при работе с небольшой программой (я использовал отладчик rhide, но отладчик в Visual C++ работает очень похоже).

Выбор отладчика

В отличие от стандартизированного языка C++, набор команд, поддерживаемый отладчиком, варьируется от производителя к производителю. К счастью, большинство отладчиков поддерживают некоторый базовый набор команд. Необходимые нам команды есть в обеих средах разработчика, описанных в этой книге. В GNU C++ и Visual C++ существует возможность вызова этих команд с помощью меню и функциональных клавши. В табл. 11.1 приведен список основных команд и клавиш их вызова.

Таблица 11.1. Команды отладчиков Microsoft Visual C++ и GNU rhide

КОМАНДА	VISUAL C++	GNU C++ (RHIDE:)
Собрать (Build)	<Shift+F8>	<F9>
Шаг с входом в функцию (Step In)	<F11 >	<F7>
Шаг без входа в функцию (Step Over)	<F10>	<F8>
Просмотр переменной (View Variable)	Только в меню	<Ctrl+F4>
Установка точки останова (Set Breakpoint)	<F9>	<Ctrl+F8>
Наблюдение; за переменной (Add watch)	Только в меню	<Ctrl+F7>
Выполнение (Go)	<F5>	<Ctrl+F9>
Просмотр экрана пользователя (View User Screen)	Щелчок в окне программы	<Alt+F5>
Перезагрузка программы (Program Reset)	<Shift+F5>	<Ctrl+F2>

Запуск тестовой программы

Лучший способ исправить ошибки в программе — пройти ее пошагово. Приведенная ниже программа содержит несколько ошибок, которые надо найти и исправить.

```
//    Concatenate — соединяет две строки и добавляет
//                  "-" между ними.
//                  (в этой версии содержатся ошибки)
#include <stdio.h>
#include <iostream.h>

void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    cout << "Эта программа соединяет две строки\n" ;
    cout << "(Эта программа содержит ошибки.)\n\n";

    // считать первую строку...
    char szStrir.gl [256];
    cout << " Введите строку #1:";
    cin.getline(szString1, 128);

    // ...теперь вторую...
    char szString2 [128];
    cout << " Введите строку #2:";
    cin.getline(szString2, 128);

    // ...присоединить " — " к концу первой...
    concatString(szString1, " — ");

    // ...теперь добавить вторую строку...
    concatString(szString1, szString2);

    // ...и показать результат
    cout << "\n" << szString1 << "\n";

    return 0;
}

// concatString - присоединяет* pszSource к окончанию
```

```
//                 строки* pszTarget
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex;
    int nSourceIndex;

    // найти конец первой строки
    while(szTarget[++nTargetIndex])
    {
    }

    // присоединяет вторую строку к концу первой
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }
}
```

Соберите и запустите программу. Когда она запросит первую строку, введите что-нибудь наподобие "это строка", а на запрос второй строки введите "ЭТО СТРОКА" (впрочем, можно писать все, что угодно).

Вместо объединения двух строк программа завершится с кодом выхода 0xff. Щелкните на OK (других вариантов у вас все равно нет). Чтобы утешить вас, отладчик откроет окно, содержащее следующую информацию:

```
Call frame traceback
Concatenate.cpp(49) in function concatString__FPcTO
Concatenate.cpp(28) in function main
in function___crt1_startup+174
```

Из этого видно, что ошибка возникла в строке 49 модуля Concatenate.cpp, которая находится в функции concatString(), вызванной в строке 28 функции main().

Строка 49 имеет вид

```
while(szTarget[++nTargetIndex])
```

А строка 28 содержит вызов функции

```
concatString(szString1, " — ") ;
```

Внешне вроде бы все в порядке и с командой в строке 49, и с вызовом функции в строке 28. Чтобы найти ошибку, понадобится отладчик.

Примечание. Хотя вы уже встречались с похожей ошибкой, я все равно настоятельно советую детально разобрать этот пример.

Пошаговое выполнение программы

Первое, что стоит сделать при поиске ошибки с помощью отладчика, — это выполнить программу в пошаговом режиме. В среде rhide выполните команду Run⇨Program Reset.

Примечание. Из табл. 11.1 видно, что в rhide эта команда выполняется с помощью клавиш <Ctrl+F2> (или клавиш <Shift+F5> в Visual C++). Больше я не буду давать подсказок, поскольку все необходимые команды содержатся в табл. 11.1. Кроме того, все команды, используемые в этой главе, доступны из выпадающих меню среды разработчика.

Команда Program Reset заставляет отладчик заново начать работу с программой (а не с того места, где вы находитесь). Никогда не вредно перезагрузить отладчик перед началом работы.

Выполните команду Step Over (пошаговое выполнение без входа в функции). Среда rhide откроет окно MS DOS, как и в нормальном режиме, однако сразу после этого переключится в окно редактирования программы, выделив первую исполняе-

мую команду. (Исполняемой называется любая команда, которая не является объявлением или комментарием. Именно исполняемая команда вызывает создание машинного кода во время компиляции.)

После запуска отладчик выполняет все команды вплоть до первой строки функции main (), а затем перехватывает управление. Теперь отладчик ожидает действий со стороны программиста.

Еще раз выполните команду Step Over — rhide вновь на секунду перейдет в окно пользователя и затем вернется в окно редактирования программы. Теперь будет выделена следующая строка. Щелкните мышью на окне пользователя (User Screen), и вы увидите выведенную во время выполнения предыдущей команды строку

```
Эта программа соединяет две строки.
```

Выполнение программы в таком режиме называется *пошаговым.* Необходимо продолжать выполнение программы в пошаговом режиме до тех пор, пока она не выдаст сообщение об ошибке. Знание команды, которая вызвала ошибку, очень поможет при ее исправлении.

Когда вы попытаетесь выполнить команду Step Over в строке с вызовом функции cin.getline(), отладчик не вернет управление среде разработчика, как обычно. Может показаться, что программа зависла, но причина в другом: отладчик не перехватывает управления, пока не завершится команда, а функция getline () не может завершиться, пока пользователь не введет строку с клавиатуры.

Введите первую строку и нажмите <Enter>. Теперь отладчик перехватит управление и остановит выполнение программы после команды cout << "Введите строку #2:". Выполните еше один шаг и введите вторую строку.



Если отладчик остановился и не возвращается в среду программиста при пошаговом выполнении, это означает, что программа ожидает какого-то события. Скорее всего, программа ожидает ввода с клавиатуры или с другого внешнего устройства.

При попытке выполнить функцию concatstring() в пошаговом режиме Step Over программа снова выполнит аварийный останов, как и ранее (рис. 11.4).

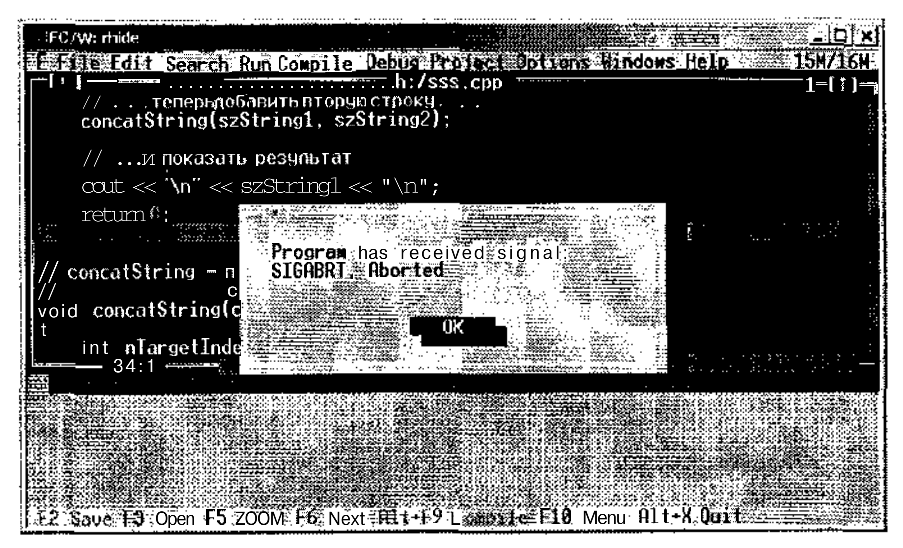


Рис. 11.4. Ошибка в функции concatstring() заставляет программу аварийно завершаться

Выведенное сообщение не поможет вам в поиске ошибки — необходимо пошагово входить в функции и выполнять их.

Пошаговый режим с входом в функции

Отладчик позволяет программисту входить в функции и выполнять их в пошаговом режиме. Эта возможность понадобится нам при поиске первой ошибки в этой программе.

Сначала следует выполнить сброс программы. Для этого перезагрузите отладчик с помощью команды Program Reset.

Затем выполняйте программу в пошаговом режиме Step Over, пока не дойдете до функции `concatString()`. После этого воспользуйтесь командой Step In (пошаговый режим с заходом в функции), и указатель перейдет на начало функции `concatString()`, как показано на рис. 11.5.

```
EFile Edit Search Run Compile Debug Project Options Windows Help     15M/16M
[•]═══════════════════════════h:/sss.cpp══════════════════════════1═[↑]
//              строки" pszTarget
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex;
    int nSourceIndex;

    // ... конец      строки
    {
    }

    // присоединяет вторую строку к концу первой
    while(szSource[nSourceIndex])
    {
══ 44:1 ══
F2 Save F3 Open F5 Zoom F6 Next F9 Compile F10 Menu Quit
```

Рис. 11.5. Команда Step In заставляет отладчик перейти на первую выполняемую строку вызываемой функции



Между командами Step In и Step Over нет никакой разницы, если только в процессе прохода не выполняется вызов функции.



Если ненамеренно выполнить команду Step In, отладчик может запросить имя файла с исходным кодом функции, о которой вы никогда не слышали. Возможно, этот исходный код находится в какой-то из библиотек. В этом случае щелкните на кнопке Cancel и попробуйте разобраться в наборе машинных инструкций (которые вряд ли много скажут даже опытным программистам). Чтобы вернуть себе после этого душевное равновесие, откройте окно редактирования программы, установите точку останова так, как описано в следующем разделе, и выполните команду Go.

А пока воспользуйтесь командой Step Over, чтобы выполнить первую исполняемую строку функции. Сразу после этого `rhide` выведет такое же сообщение об ошибке, как и раньше.

Теперь достоверно известно, что ошибка находится в операторе while и что именно она вызывает аварийное завершение программы. Чтобы узнать, что именно происходит, следует остановить программу непосредственно перед выполнением оператора while и разобраться в ней более детально.

Использование точек останова

Пошаговый режим хорош только на этапе предварительного поиска ошибки. После того как стало известно, в каком месте программы возникла ошибка, удобнее всего воспользоваться возможностью отладчика, называемой *точкой останова.*

Чтобы увидеть, как работает точка останова, вновь сбросьте отладчик с помощью команды Program Reset. Мы могли бы снова добраться до оператора while в пошаговом режиме, как это было сделано ранее, однако постоянно выполнять большие программы в пошаговом режиме было бы слишком утомительно. Вместо этого можно установить точку останова. Поместите курсор на операторе while и выполните команду Set breakpoint. Редактор выделит эту строку красным цветом, как показано на рис. 11.6.

```
{
    int nTargetIndex;
    int nSourceIndex;

    // найти конец первой строки
    while(szTarget[++nTargetIndex])
    {
    }

    // присоединяет вторую строку к концу первой
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
```

Рис. 11.6. Строка с точкой останова выделена красным цветом

Точка останова заставляет программу работать в нормальном режиме до точки, в которой нужно остановить программу. После этого отладчик перехватывает управление и останавливает программу. Точки останова очень удобны, если вы уже знаете, где искать ошибку, или если просто возникла необходимость выполнить программу до определенного места.

После установки точки останова выполните команду Go. Программа будет выполняться нормально до тех пор, пока не дойдет до оператора while. После этого она передаст управление отладчику.

Осталось только узнать, что же вызывает ошибку.

Просмотр и редактирование переменных

Нет смысла вновь выполнять оператор while, поскольку точно известно, что программа будет прервана в результате ошибки. Чтобы понять, почему возникает ошибка, необходимо получить дополнительную информацию. Например, не помешало бы узнать, что находится в переменной nTargetIndex непосредственно перед выполнением цикла while.

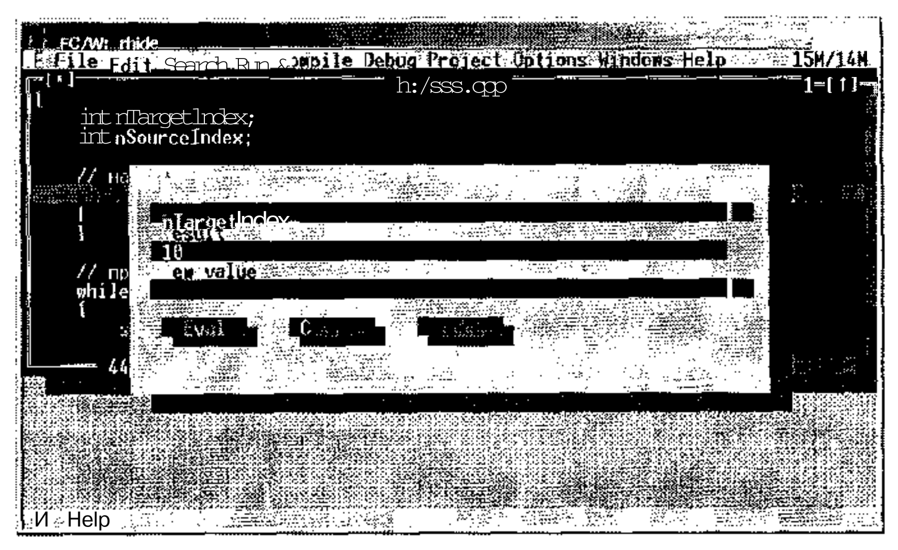


Рис. 11.7. Отладчик позволяет программисту просматривать и изменять содержимоепеременных

Сначала дважды щелкните мышью на имени переменной nTargetIndex, а после этого вызовите команду отладчика View Variable. При этом появится окно с именем этой переменной в первой строке. Щелкните на кнопке Eval, чтобы узнать текущее значение переменной. Результат этих действий показан на рис. 11.7.

Если еще раз внимательно просмотреть исходную программу, становится понятно, что в процессе ее выполнения не были инициализированы переменные nTargetIndex и nSourceIndex. Чтобы убедиться в этом, введите в строке ввода нового значения (New Value) значение 0 и щелкните на кнопке OK (то же самое нужно сделать и для переменной nSourceIndex). После этого можно закрыть окно просмотра переменных и выполнить команду Step Over.

Теперь, после инициализации этих переменных, программа уже не выдаст сообщения об ошибке. Каждое следующее выполнение команд Step Over или Step In вызовет повтор цикла while. Поскольку тело цикла не содержит никаких команд, курсор снова вернется на оператор while. При этом в ходе каждого выполнения nTargetIndex будет увеличиваться на 1.

Чтобы не открывать окно просмотра переменной nTargetIndex во время каждого выполнения цикла, дважды щелкните на nTargetIndex и вызовите команду Add Watch. Появится окно с именем этой переменной и ее значением в правой части. После этого выполните команду Step In еше несколько раз. При каждом повторении nTargetIndex будет увеличиваться на 1, а после нескольких циклов произойдет переход на следующую за циклом строку.

Установите точку останова на строку, содержащую закрывающую фигурную скобку функции concatString, и выполните команду Go. Программа остановится непосредственно перед выходом из функции.

Чтобы проверить содержимое созданной этой функцией строки, дважды щелкните на переменной szTarget и выполните команду View Variable. Результат приведен на рис. 11.8.



Число 0x68298 является адресом строки в памяти. Эта информация может пригодиться при отслеживании указателей. Адреса могут очень пригодиться при работе со связанными списками.

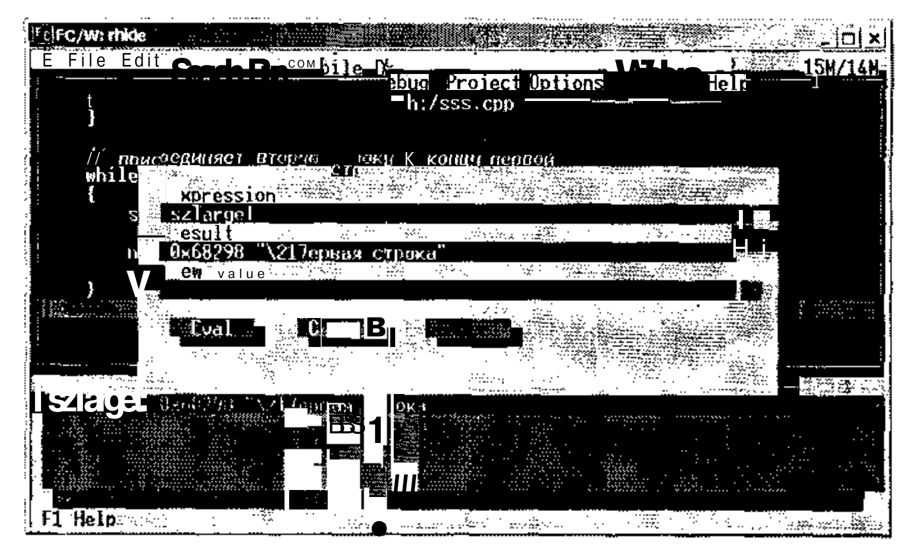


Рис. 11.8. Даже после инициализации переменных результат соединения строк оказывается неверным



Дописывание символов после символа окончания строки (нулевого символа) или отсутствие символа окончания строки — две самых распространенных ошибки при работе со строками.

Теперь, когда известно, где находятся обе ошибки, было бы неплохо исправить их в исходной программе (пока мы не забыли, где они находятся). Вызовите команду Program Reset и исправьте функцию `concatString` так, как показано в следующем листинге:

```
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex = 0;
    int nSourceIndex = 0;

    // найдем конец первой строки
    while(szTarget[++nTargetIndex])
    {
    }

    // присоединяем вторую строку к концу первой
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // завершаем конечную строку
    szTarget[nTargetIndex] = '\0';
}
```

То, что было найдено несколько ошибок, еше не значит, что в программе нет других "жучков". Необходимо довести процесс отладки до конца. Установите просмотр переменных `nTargetIndex` и `nSourceIndex` и самостоятельно проверьте их содержимое во время выполнения следующего цикла.



Вам действительно необходимо сделать это самостоятельно. Только тогда вы поймете, какое эстетическое удовольствие можно получить, глядя на то, как от цикла к циклу одна строка растет, а другая одновременно уменьшается.

Чтобы убедиться, что все работает правильно, удалите все точки останова и запустите программу с помощью команды Go. Приведенный ниже вывод программы выглядит правильно.

```
Эта программа соединяет две строки
 (Эта программа не содержит ошибок.)

Введите строку #1:эта строка
Введите строку #2:ЭТА СТРОКА

эта строка - ЭТА СТРОКА
```

Мои поздравления новоиспеченным экспертам по отладке программ!

Первая программа BUDGET

Главы, составляющие первые две части книги, позволяют вам писать собственные (уже нетривиальные) программы. Именно такова приведенная ниже программа BUDGET.

Эта программа будет представлена в книге еще не раз. И в каждой версии будут использованы новые возможности, изученные в предыдущих главах. Таким образом, вы увидите, как применить возможности языка, описанные ранее. Текущий вариант программы использует "функциональные" возможности (т.е. возможности функций) языка C++.

Программа BUDGET моделирует простейший банковский счет (в очень упрощенном виде). Эта программа будет выполнять такие действия:

- ✓ создавать один или несколько банковских счетов;
- ✓ присваивать уникальный номер каждому счету;
- ✓ работать со счетом — создание депозита и снятие денег;
- ✓ выводить окончательный баланс всех счетов, после того как пользователь решит выйти из программы.

Эта версия программы будет следить за тем, чтобы баланс не был меньше нуля (банк может быть дружественным по отношению к вам, но не до такой же степени!). Набор правил будет разрастаться с развитием программы в следующих частях книги.

Программа BUDGET приведена ниже.

```
// BUDGET1.CPP — "Функциональная" программа бюджета
#include <iostream.h>
#include <stdio.h>

// максимальное количество счетов
const int maxAccounts = 10;

// информация о счете
unsigned accountNumber[maxAccounts];
double   balance[maxAccounts];

// Прототипы функций
void process (unsigned& accountNurnber,
              doubles   balance);
```

```
void init(unsigned& accountNumber,
          doubles   balance);

// main — собирает начальные входные данные
//и выводит конечные суммы
int main(int nArg, char* pszArgs[])
{
   // ожидаем ввод
   int noAccounts = 0;  // количества создаваемых счетов

   // не создаем счетов больше максимального количества
   while (noAccounts < maxAccounts)
   {
       char transactionType;
       cout << "Нажмите C для продолжения, X для выхода:";
       cin  >> transactionType;

       // выйти, если пользователь ввел X
       if (transactionType == 'x' ||
           transactionType == 'X')
       1
           break;
       }

       // если пользователь ввел C...
       if (transactionType == 'c' ||
           transactionType == 'C')
       i
          // ...начать создание нового счета...
          init(accountNumber[noAccounts],
               balance[noAccounts]);

          // ...и ввести информацию о проводке
          process(accountNumber[noAccounts],
                  balance[noAccounts]);

          // подсчитать количество счетов
          noAccounts++;
       }
   }

   // показать сумму
   // для каждого счета в отдельности
   double total = 0;
   cout << "Информация о счетах:\n";
   for (int i = 0; i < noAccounts; i++)
   i
      ccut << "Баланс счета "
           << accountNumber[i]
           << " = "
           << balance[i]
           << "\n";

      // подсчитать сумму по Есем счетам
      total += balance[i];
   }

   // вывести сумму по всем счетам
```

```
   cout << "Баланс по всем счетам = "
        << total
        << "\n";

   return 0;
}

// init — инициализирует счет, считав
//        его номер к обнулив баланс
void init(unsigned& accountNumber,
          doubles   balance)
{
   cout << "Введите номер счета:";
   cin  >> accountNumber;
   balance = 0.0;
}

// process — изменяет баланс счета
//           в соответствии с транзакцией пользователя
void process (unsigned& accountNumber,
              doubles   balance)
{
   cout << "Введите положительную сумму вклада\n"
        << "или отрицательную сумму для снятия со счета\n";
   double transaction;
   do
   (
      cout << ":";
      cin  >> transaction;

      // это вклад?
      if (transaction > 0)
      {
         balance += transaction;
      }

      // или снятие?
      if (transaction < 0)
      {
         // снятие
         transaction = -transaction;
         if (balance < transaction)
         {
            cout << "Недостаточно денег на счете: баланс "
                 << balance
                 << ", сумма транзакции "
                 << transaction
                 << "\n";
         }
         else
         {
            balance -= transaction;
         }
      }
   } while (transaction != 0) ;
}
```

Чтобы продемонстрировать эту программу в действии, я ввел следующие числа (вывод программы обозначен нормальным шрифтом, мой ввод — жирным):

```
Нажмите С для продолжения или X для отмены:C
Введите номер счета:1234
Введите положительную сумму вклада
или отрицательную сумму для снятия со счета
:200
:-100
:-200
Недостаточно денег на счете: баланс 100, сумма транзакции 200
:0
Нажмите С для продолжения или X для отмены:C
Введите номер счета:2345
Введите положительную сумму вклада
или отрицательную сумму для снятия со счета
:200
:-50
:-50
:-50
:0
Нажмите С для продолжения иди X для отмены:X
Баланс счета 1234 = 100
Баланс счета 2345 = 50
Баланс го всем счетам = 150
```

Стиль программирования

Вы, наверное, заметили, что я пытался быть последовательным в отступах и в именовании переменных.

Наша голова имеет ограниченную "производительность". И эту производительность следует направлять на создание работающих программ, а не на расшифровку уже написанных, которые невозможно прочитать из-за плохого оформления.

Важно, чтобы вы научились правильно именовать свои переменные, корректно располагать скобки и выполнять многое другое, что составляет стиль программирования. Разработав стиль программирования, придерживайтесь его, и он войдет в привычку. Однажды вы заметите, что пишете программы быстрее, а их чтение не вызывает затруднений.

Это особенно важно, когда над проектом работают несколько программистов, поскольку правильный стиль помогает избежать проблем, возникших у строителей Вавилонской башни. Кроме того, я бы настоятельно советовал тщательно разбираться в каждом сообщении об ошибке или предупреждении компилятора. Даже если считать, что предупреждение — это еще не ошибка, то зачем дожидаться, пока оно превратится в ошибку? Тем более, что, если оно такое простое, каким кажется, разобраться в нем и устранить его не составит труда. В большинстве случаев предупреждения вызываются ошибочным стилем программирования, который лучше исправить.

Одни говорят, что недосмотры — это их личное дело, другие же считают, что это лишняя трата времени. Если вы так думаете, то просто представьте себе, как обидно будет обнаружить ошибку, о которой компилятор предупреждал вас давным-давно.

Разберемся в том, как работает BUDGET. В этой программе было создано два массива, один из которых содержит номера счетов, а второй — балансы. Эти массивы синхронизированы таким образом, что элемент `balance[n]` содержит баланс счета с номером из `accountNumber[n]`, независимо от значения n. В связи с ограничением

длины массива количество счетов, содержащихся в программе, не может превышать MAXACCOUNTS.

Главная программа разделена на две части: первая отвечает за сбор информации; в ней происходит считывание размеров вкладов, снятие денег и запись результата, а вторая — за вывод информации. Фрагмент, отвечающий за сбор информации, организован в виде цикла, в котором счета обрабатываются каждый в отдельности. В начале цикла пользователю предлагается ввести с для продолжения работы и X — для завершения. Если был введен символ X, происходит выход из цикла и переход во вторую часть main ().

Программа выходит из цикла, если количество созданных счетов достигло MAXACCOUNTS, независимо от того, был или нет введен x.

Обратите внимание, что происходит проверка введенного символа на равенство как 'X', так и 'x' — ведь в отличие от компьютера человек может не обратить внимания на регистр вводимых символов.

Если пользователь ввел ‘C’, то управление передается функции init(), которая создает счет и заполняет его необходимой информацией. После этого функция process (> добавляет в счет информацию о транзакции.

Аргументами функций init() и process {} являются указатели, так что эти функции могут изменять значения своих аргументов. В противном случае обновленная информация о счете была бы утрачена по окончании работы функций.

После завершения создания счетов управление переходит блоку, отвечающему за вывод итоговых результатов. В этом блоке происходит считывание каждого счета и вывод баланса каждого из них. В конце выводится общая сумма.

Функция init () создает новый счет после приглашения ввести его номер и обнуляет создаваемый счет.

Очень важно не забыть проинициализировать новый элемент. Нулевой баланс счета лучше непредсказуемого значения (например, отрицательного).

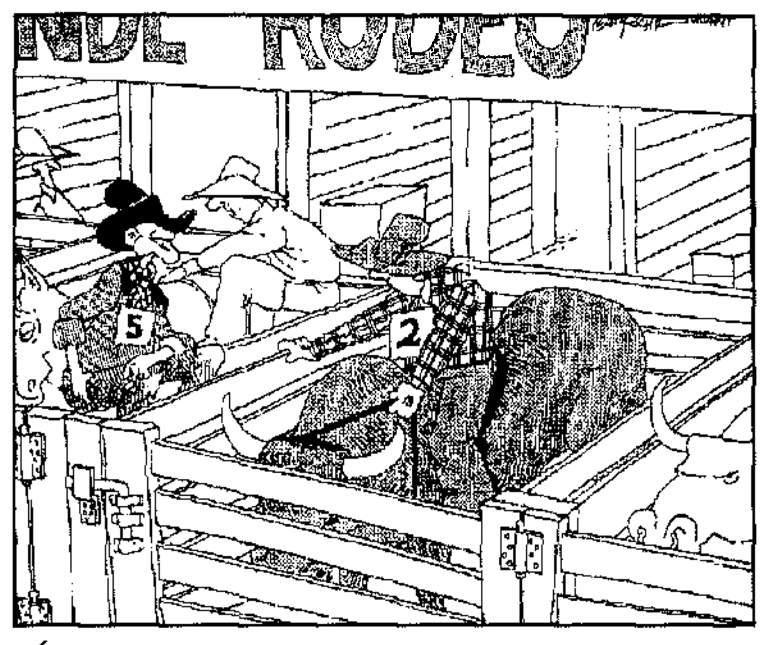
Функция process!) использует цикл для ввода каждой новой транзакции. Положительные значения считаются вкладом, а отрицательные — снятием со счета. Для завершения работы со счетом достаточно ввести ноль.

Значение 0 используется программой в качестве флага. Это хотя и довольно распространенный, но не самый хороший метод. Я использовал его в этой программе только потому, что он помогает сохранить довольно много места.

Существует множество способов улучшить программу BUDGET. ИХ ПОИСК станет основой для последующего изучения языка C++. В конце книги вы увидите, как эта программа превратится в полнофункциональную объектно-ориентированную программу C++.

Часть III

"Классическое" программирование



Сэнди, поосторожнее—этот парень неделю тестировал программу с паршивой документацией, так что теперь он так и хочет на ком-то отыграться!

В этой части...

Основным отличием C++ от других языков является возможность объектно-ориентированного программирования. Термин объектно-ориентированный *— один из самых популярных в современном компьютерном мире. Языки программирования, редакторы и базы данных — буквально все претендуют на звание объектно-ориентированных. Иногда так оно и есть, но часто такое определение дается* **исключительно** *в рекламных целях. Так что же такое объектно-ориентированное программирование? Прочитайте эту часть книги.*

Глава 12

Знакомство с объектно-ориентированным программированием

В этой главе...

- ✓ Микроволновые печи и уровни абстракции
- ✓ Классифицирование микроволновых печей
- ✓ Зачем нужна классификация

Что такое объектно-ориентированное программирование вообще? Объектно-ориентированное программирование, или ООП, базируется на двух принципах, которые нам известны еще с младенческого возраста: абстракция и классификация. Чтобы пояснить, что имеется в виду, я расскажу вам одну историю.

Микроволновые печи и уровни абстракции

Когда мы с сыном смотрим футбол, я подчас испытываю непреодолимую тягу к вредным для здоровья, но таким вкусным мексиканским блюдам. Я бросаю на тарелку чипсы, бобы, сыр, приправы и пять минут зажариваю эту массу в микроволновой печи.

Для того чтобы воспользоваться печью, я открываю ее дверцу, забрасываю внутрь полуфабрикат и нажимаю несколько кнопок на передней панели. Через пару минут блюдо готово (я стараюсь не стоять перед печью, чтобы мои глаза не начали светиться в темноте).

Обратите внимание на то, чего я *не делом,* используя свою микроволновую печь.

- ✓ Не переключал и не изменял ничего внутри печи. Чтобы установить для нее рабочий режим, существует интерфейс — лицевая панель с кнопками и небольшой индикатор времени; это все, что мне нужно.
- ✓ Не перепрограммировал процессор внутри печи, даже если прошлый раз готовилось абсолютно другое блюдо.
- ✓ Не смотрел внутрь печи.
- ✓ Не задумывался бы над ее внутренним устройством во время приготовления блюд даже в том случае, если бы был главным инженером по производству печей и знал о них все, включая каждую программу.



Это не пространные рассуждения. В повседневной жизни нас постоянно преследуют стрессы. Чтобы уменьшить их число, мы начинаем обращать внимание только на события определенного масштаба. В объектно-ориентированном программировании уровень детализации, на котором вы работаете, называется *уровнем абстракции.* Например, чтобы объяснить этот термин, я *абстрагируюсь* от подробностей внутреннего устройства микроволновой печи.

Во время приготовления блюда я смотрел на микроволновую печь просто как на железный ящик. И пока я управляю печью с помощью интерфейса, я не могу ее сломать, "подвесить" или, что еще хуже, превратить свое блюдо в угли.

Приготовление блюд с помощью функций

Представьте себе, что я попросил бы своего сына написать алгоритм приготовления мною закусок. Поняв наконец, чего я от него добиваюсь, он бы, наверное, написал что-то вроде "открыть банку бобов, натереть сыра, посыпать перцем" и т.д. Когда дело дошло бы непосредственно до приготовления в печи, он в лучшем случае написал бы нечто подобное: "готовить в микроволновой печи пять минут".

Этот рецепт прост и верен. Но с помощью такого алгоритма "функциональный" программист не сможет написать программу приготовления закусок. Программисты, работающие с функциями, живут в мире, лишенном таких объектов, как микроволновая печь и прочие удобства. Они заботятся о последовательности операций в функциях. В "функциональном" решении проблемы закусок управление будет передано от моих пальцев кнопкам передней панели, а затем внутрь печи. После этого программе придется решать, на какое время включать печь и когда следует включить звуковой сигнал готовности.

При таком подходе очень трудно отвлечься от сложностей внутреннего устройства печи. В этом мире нет объектов, за которые можно спрятать всю присущую микроволновой печи сложность.

Приготовление "объектно-ориентированных" блюд

Применяя объектно-ориентированный подход к приготовлению блюд, я первым делом определю объекты, используемые в задаче: сыр, бобы, чипсы и микроволновая печь. После этого я начинаю моделировать эти объекты в программе, не задумываясь над деталями их использования.

При этом я работаю (и думаю) на уровне базовых объектов. Я должен думать о том, как приготовить блюдо, не волнуясь о деталях работы микроволновой печи — над этим уже подумали ее создатели (которым нет дела до моих любимых блюд).

После создания и проверки всех необходимых объектов можно переключиться на следующий уровень абстракции. Теперь я начинаю думать на уровне процесса приготовления закуски, не отвлекаясь на отдельные куски сыра или банки бобов. При таком подходе я легко переведу рецепт моего сына на язык C++.

Классифицирование микроволновых печей

В концепции уровней абстракции очень важной частью является классификация. Если бы я спросил моего сына: "Что такое микроволновая печь?" — он бы наверняка ответил: "Это печь, которая...". Если бы затем я спросил: "А что такое печь?" — он бы ответил что-то вроде: "Ну, это кухонный прибор, который...". (Если бы я попытался выяснить, что такое кухонный прибор, он наверняка бы спросил, почему я задаю так много дурацких вопросов.)

Из ответов моего сына становится ясно, что он видит нашу печь как один из экземпляров вещей, которые называются микроволновыми печами. Кроме того, печь является подразделом духовок, а духовки относятся к типу кухонных приборов.



В ООП моя микроволновая печь является *экземпляром* класса микроволновых печей. Класс микроволновых печей является подклассом печей, который, в свою очередь, является подклассом кухонных приборов.

Люди склонны заниматься классификацией. Все вокруг увешано ярлыками. Мы делаем все, для того чтобы уменьшить количество вещей, которые надо запомнить. Вспомните, например, когда вы первый раз увидели "Пежо" или "Рено". Возможно, в рекламе и говорилось, что это суперавтомобиль, но мы-то с вами знаем, что это не так. Это ведь просто машина. Она имеет все свойства, которыми обладает автомобиль. У нее есть руль, колеса, сиденья, мотор, тормоза и т.д. Могу поспорить, что я смог бы даже водить такую штуку без инструкции.

Я не буду тратить место в книге на описание того, чем этот автомобиль похож на другие. Мне нужно знать лишь то, что это "машина, которая...", и то, чем она отличается от других машин (например, ценой). Теперь можно двигаться дальше. Легковые машины являются таким же подклассом колесных транспортных средств, как грузовики и пикапы. При этом колесные транспортные средства входят в состав транспортных средств наравне с кораблями и самолетами.

Зачем нужна классификация

Зачем вообще классифицировать? Ведь это влечет за собой массу трудностей. Тем более, что у нас уже есть готовый механизм функций. Зачем же что-то менять?

Иногда может показаться, что легче разработать и создать микроволновую печь специально для некоторого блюда и не строить универсальный прибор на все случаи жизни. Тогда на лицевую панель не надо будет помещать никаких кнопок, кроме кнопки СТАРТ. Блюдо всегда готовилось бы одинаковое время, и можно было бы избавиться от всех этих бесполезных кнопок типа РАЗМОРОЗКА или ТЕМПЕРАТУРА ПРИГОТОВЛЕНИЯ. Все, что требовалось бы от такой печи, — это чтобы в нее помешалась одна тарелка с полуфабрикатом. Да, но что же тогда получится? Ведь при этом один кубический метр пространства использовался бы для приготовления всего одной тарелки закуски!

Чтобы сэкономить место, можно освободиться от этой глупой концепции — "микроволновая печь". Для приготовления закуски хватит и внутренностей печи. Тогда в инструкции достаточно написать примерно следующее: "Поместите полуфабрикат в ящик. Соедините красный и черный провод. Установите на трубе излучателя напряжение в 3000 вольт. Должен появиться негромкий гул. Постарайтесь не стоять близко к установке, если вы хотите иметь детей". Простая и понятная инструкция!

Но такой функциональный подход создает некоторые проблемы.

- ✓ **Слишком сложно.** Я не хочу, чтобы фрагменты микроволновой печи перемешивались с фрагментами закуски при разработке программы. Но поскольку при данном подходе нельзя создавать объекты и упрощать написание, работая с каждым из них в отдельности, приходится держать в голове все нюансы каждого объекта одновременно.
- ✓ **Не гибко.** Когда-нибудь мне потребуется поменять свою микроволновую печь на печь другого типа. Я смогу это сделать без проблем, если интерфейс печи можно будет оставить старым. Без четко очерченных областей действия, а также без разделения интерфейса и внутреннего содержимого становится крайне трудно убрать старый объект и поставить на его место новый.
- ✓ **Невозможно использовать повторно.** Печи делаются для приготовления разных блюд. Мне, например, не хочется создавать новую печь всякий раз, когда требуется приготовить новое блюдо. Если задача уже решена, неплохо использовать ее решение и в других программах.

Глава 13

Классы в С++

В этой главе...

✓ Введение в классы

Очень часто программы имеют дело с совокупностями данных: имя, должность, табельный номер и т.д. Каждая отдельная составляющая не описывает человека, смысл имеет только вся вместе взятая информация. Простая структура, такая как массив, прекрасно подходит для хранения отдельных значений, однако совершенно непригодна для хранения совокупности данных разных типов. Таким образом, массив недостаточен для хранения *комплексной* информации.

По причинам, которые вскоре станут понятными, я буду называть такие совокупности информации *объектами.* Микроволновая печь — объект. Вы также объект (и я тоже, хотя уже и не так уверен в этом). Ваше имя, должность и номер кредитной карты, содержащиеся в базе данных, тоже являются объектом.

Введение в классы

Для хранения разнотипной информации о физическом объекте нужна специальная структура. В нашем простейшем примере эта структура должна содержать поля имени, фамилии и номера кредитной карты.

В С++ структура, которая может объединить несколько разнотипных переменных в одном объекте, называется *классом.*

Формат класса

Класс, описывающий объект, который может содержать имя и номер кредитной карты, может быть создан так;

```
// Класс dataset
class NameDataSet
{
    public:
        char firstName[128];
        char lastName [128];
        int  creditCard;
};
// экземпляр класса dataset
NameDataSet nds;
```

Объявление класса начинается с ключевого слова class, после которого идет имя класса и пара фигурных скобок, открывающих и закрывающих тело класса.

Можно использовать альтернативное ключевое слово struct, которое полностью идентично class, с предполагаемым использованием объявлений public.

После открывающей скобки находится ключевое слово public. (Не спрашивайте меня сейчас, что оно значит, — я объясню его значение немного позже. В следующих главах поясняются разные ключевые слова, такие как public или private. А до тех пор пока я не сделаю private публичным, значение pubiic останется приватным :-).)

После ключевого слова public идет описание полей класса. Как видно из листинга, класс NaneDaraSet содержит поля имени, фамилии и номера кредитной карты. Первые два поля являются символьными массивами, а третье имеет тип int (будем считать, что это и есть номер кредитной карты).



Объявление класса содержит поля данных, необходимые для описания единого объекта.

В последней строке этого фрагмента объявляется переменная nds, которая имеет тип NameDataSet. Таким образом, nds представляет собой запись, описывающую отдельного человека.

Говорят, что nds является *экземпляром* класса NameDataSet и что мы создали этот экземпляр, *реализовав* класс NameDataSet. Мы говорим, что поля firstName и остальные являются *членами,* или *свойствами* класса. Эти глупые слова будут использоваться и в дальнейшем в рассуждениях о классах.

Обращение к членам класса

Обратиться к членам класса можно так:

```
NameDataSet nds;
nds.creditCard = 1C;
cin >> nds.firstName;
cin >> nds.lastName;
```

Здесь nds — экземпляр класса NameDataSet (или отдельный объект типа NameDataSet); целочисленная переменная nds.creditCard— свойство объекта nds; член nds.crecitCard имеет тип int, тогда как другой член этого объекта, nds . firstName, имеет тип char [].

Если не пользоваться компьютерным сленгом, приведенный пример можно объяснить так: в этом фрагменте программы происходит объявление объекта nds, который затем будет использован для описания покупателя. По каким-то соображениям программа присваивает этому человеку кредитный номер 10 (понятно, что номер фиктивный — я ведь не собираюсь распространять номера своих кредитных карт!).

Затем программа считывает имя и фамилию из стандартного ввода.

Теперь программа может работать с объектом nds как с единым целым, не обращаясь к его отдельным частям, пока в этом не возникает необходимость.

```
int getData(NameDataSet& nds)
{
    cout << "\nВведите имя: ";
    cin  >> nds.firstName;

    if (stricmp(nds.firstName, "exit") == 0)
    {
        return 0;
    }

    cout << "Введите фамилию:";
    cin  >> nds . lastName;

    cout << "Введите номер кредитной карты:";
    cin  >> nds.creditCard;

    return 1;
}
```

```
// displayData — выводит содержимое множества данных
void displayData(NameDataSet& nds)
{
    cout << nds.firstName
         << " "
         << nds.lastName
         «  "/"
         << nds.creditCard
         «  "\n";
}

int main(int nArg, char* pszArgs[])
{
    const int MAX = 25;
    // создать 25 объектов типа NameDataSet
    NameDataSet nds[MAX];

    // считывает имена, фамилии и номера
    // кредитных карт
    coat << "Считывает имя/номер карть:\п"
         << "Введите в поле имени 'exit'\n"
         « "для выхода из программы\n";
    int index = 0;
    while (getData(nds[index}) && index < MAX)
    {
        index++;
    }

    cout « "\nЭлементы:\n";
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i});
    }
    return 0;
}
```

Пример программы

Приведенный ниже пример демонстрирует использование класса NameDataSet,

```
// DataSet - записывает соответствующую информацию
//              в массиве объектов
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet — содержит имя клиента
//и номер кредитной карты
class NameDataSet
{
    public:
        char firstName[128];
        char lastName [128];
        int  creditCard;
};

int getData(NameDataSet& nds)
{
    cout << "\nВведите имя: ";
```

```
    cin  >> nds.firstName;

    if (stricmp(nds.firstName, "exit") == 0)
    {
        return 0;
    }

    cout << "Введите фамилию: ";
    cin  >> nds.lastName;

    cout << "Введите номер кредитной карты: ";
    cin   >> nds.creditCard;

    return 1;
}

// displayData — выводит содержимое
// одного элемента NameDataSet
void displayData(NameDataSet& nds)
{
    cout << nds.firstName
         << " "
         << nds.lastName
         << "/"
         << nds.creditCard
         << "\n";
}

int main(int nArg, char* pszArgs[])
{
    const int MAX = 25;
    // создать 25 объектов типа NameDataSet
    NameDataSet nds[MAX];

    // считывает имена, фамилии и номера
    // кредитных карт
    cout << "Считывает имя/номер карты\n"
         << "Введите 'exit' в поле имени для выхода\n";
    int index = 0;
    while (getData(nds[index]) && index < MAX)
    {
        index++;
    }

    cout << "\nЭлементы:\n";
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i]);
    }
    return 0;
}
```

В функции main() создается массив из 25 объектов класса NameDataSet, после чего программа приглашает пользователя ввести необходимую информацию. Затем в теле цикла while происходит вызов функции getData(), которая ожидает ввода с клавиатуры содержимого элементов массива. Цикл прерывается, если getData() возврашает 0 или если количество заполненных объектов достигло максимального значения (в данном случае — 25). После этого созданные объекты передаются функции displayData, которая выводит их на экран.

Функция getData() принимает аргумент типа NameDataSet, которому внутри функции присваивается имя nds. Пока что не обращайте внимания на символ "&" — о нем речь пойдет в главе 15, "Создание указателей на объекты".

Внутри функции getDataO происходит считывание строки из устройства стандартного ввода с последующей его записью в член firstName. Если stricmp() находит, что введенная строка— "exit", функция getData() возвращает С в функцию main (), сигнализируя, что пора выходить из цикла ввода информации. (Функция stricmpO сравнивает строки, не обращая внимания на регистр. Строки "EXIT", "exit" и другие считаются идентичными.) Если введена не строка "exit", функция считывает из стандартного ввода фамилию и номер кредитной карты и записывает их в объект nds.

Функция displayData () выводит на дисплей все члены объекта nds.

Результат работы этой программы будет выглядеть так:

```
Считывает имя/номер карты
Введите 'exit' в поле имени для выхода
Введите имя: Stephen
Введите фамилию: Davis
Введите номер кредитной карты: 123456
Введите имя: Marshall
Введите фамилию: Smith
Введите номер кредитной карты: 567890
Введите имя: exit
Элементы:
Stephen Davis/123456
Marshall Smith/567890
```

Вывод программы начинается с пояснения, как с ней работать. В первой строке я ввел свое имя (видите, какой я скромный!). Поскольку меня не зовут exit, программа продолжает выполнение. Далее я ввел свою фамилию и номер кредитной карты. Следующим элементом массива я ввел имя Marshall Smith и номер его кредитной карты. Затем я ввел строку exit и таким образом прервал цикл заполнения объектов. Как видите, эта программа не делает ничего, кроме вывода только что введенной информации.

Глава 14

Работа с классами

В этой главе...

- ✓ Активизация объектов
- ✓ Добавление функции-члена
- ✓ Вызов функций-членов
- ✓ Доступ к членам из функции-члена
- ✓ Разрешение области видимости
- ✓ Определение функции-члена
- ✓ Определение функций-членов вне класса
- ✓ Перегрузка функций-членов

Программисты используют классы для объединения взаимосвязанных данных в один объект. Приведенный ниже класс Savings объединяет в себе баланс и уникальный номер счета.

```
class Savings
{
public:
    unsigned accountNumber;
    float balance;
};
```

Каждый экземпляр класса Savings содержит одинаковые элементы:

```
void fn(void)
(
    Savings a;
    Savings b;
    a.accountNumber = 1;  // этот счет не тот же, что и...
    b.accountNumber = 2;  // ... этот
}
```

Переменная `a.accountNumber` отличается от переменной b. accountNumber. Эти переменные различаются между собой так же, как баланс моего банковского счета отличается от вашего (хотя они оба называются балансами).

Активизация объектов

Классы используются для моделирования реально существующих объектов. Чем ближе объекты C++ к реальному миру, тем проще с ними работать в программах. На словах это звучит довольно просто, однако существующий сейчас класс `Savings` не предпринимает ничего, чтобы хоть в чем-то походить на настоящий банковский счет.

Моделирование реальных объектов

Реальные объекты имеют свойства-данные, например номера счетов и балансы. Но кроме этого, реальные объекты могут выполнять действия: микроволновые печи готовят, сберегательный счет начисляет проценты, полицейский выписывает штраф и т.д.

Функционально ориентированные программы выполняют все необходимые действия с помощью функций. Программа на C++ может вызвать функцию strcmp() для сравнения двух строк или функцию getLine() для ввода строки. В главе 26, "Использование потоков ввода-вывода", будет показано, что даже операторы работы с потоками ввода-вывода (cin >> и cout <<) являются не чем иным, как особым видом вызова функции.

Для выполнения действий классу Savings необходимы собственные активные свойства:

```
class Savings
{
public:
    unsigned accountNumber;
    float balance;
    unsigned deposit(unsigned amount)
    {
        balance += amount;
        return balance;
    }
};
```

В приведенном примере помимо номера и баланса счета в класс Savings добавлена функция deposit (). Теперь класс Savings может самостоятельно управлять своим состоянием. Так же, как класс MicrowaveOven (микроволновая печь) содержит функцию cook () (готовить), класс Savings содержит функцию deposit ().

Функции, определенные в классе, называются *функциями-членами.*

Зачем нужны функции-члены

Почему мы должны возиться с функциями-членами? Что плохого в таком фрагменте:

```
class Savings
{
public:
    unsigned accountNumber;
    float balance;
};
unsigned deposit(Savings& s, unsigned amount)
{
    s.balance += amount;
    return s.balance;
}
```

Еще раз напомню: пока что не обращайте внимания на символ "&" — его смысл станет понятен позже.

В этом фрагменте deposit() является функцией "вклада на счет". Эта функция поддержки реализована в виде внешней функции, которая выполняет необходимые действия с экземпляром класса Savings. Конечно, такой подход имеет право на существование, но он нарушает наши правила объектно-ориентированного программирования.

Микроволновая печь имеет свои внутренние компоненты, которые "знают", как разморозить и приготовить продукты или сделать картошку хрустящей. Данные-члены класса схожи с элементами микроволновой печи, а функции-члены — с программами приготовления.

Когда я делаю закуску, я не должен начинать приготовление с подключения внутренних элементов микроволновой печи. И я хочу, чтобы мои классы работали так же, т.е. чтобы они без всякого внешнего вмешательства знали, как управлять своими "внутренними органами". Конечно, такие функции-члены класса Savings, как deposit (), могут быть реализованы и в виде внешних функций. Можно даже расположить все функции, необходимые для работы со счетами, в одном месте файла. Микроволновую печь можно заставить работать, соединив необходимые провода внутри нее, но я не хочу, чтобы мои классы (или моя микроволновая печь) работали таким образом. Я хочу иметь класс Savings, который буду использовать в своей банковской программе, не задумываясь над тем, какова его рабочая "кухня".

Добавление функции-члена

Эта процедура включает два аспекта: создание функции-члена и ее именование (звучит довольно глупо, не правда ли?).

Создание функции-члена

Чтобы продемонстрировать работу с функциями-членами, начнем с определения класса Student следующим образом:

```
class Student
{
public:
    int   semesterHours;
    float gpa;

    // добавить пройденный курс к записи
    float addCourse(int hours, float grade)
    {
        // Вычисляем среднюю оценку с учетом
        // времени различных курсов
        float weightedGPA;
        weightedGPA = semesterHours * gpa;

    // добавить новый курс
    semesterHours  += hours;
    weightedGPA  += grade * hours;
    gpa = weightedGPA / semesterHours;

    // вернуть новую оценку
    return gpa;
    }
};
```

Функция addCourse (int, float) является функцией-членом класса Student. По сути, это такое же свойство класса Student, как и свойства semesterHours и gpa.

Для функций или переменных в программе, которые не являются членом какого-либо класса, нет специального названия, однако в этой книге я буду называть переменные или функции *не членами* класса, если они не были явно описаны в составе какого-либо класса.



По историческим причинам функции-члены называют также *методами*. Такое название имеет смысл в других объектно-ориентированных языках программирования, но бессмысленно в C++. Несмотря на это, термин приобрел некоторую популярность и среди программистов на C++, наверное, поскольку его проще выговорить, чем выражение "функция-член" (то, что это звучит гораздо внушительнее, никого не волнует). Так что если во время вечеринки ваши друзья начнут сыпать словечками вроде "методы класса", просто мысленно замените "методы" выражением "функции-члены", и все встанет на свои места. Поскольку термин "метод" смысла в C++ не имеет, я не буду использовать его в этой книге.

Именование членов класса

Функция-член во многом похожа на члена семьи. Полное имя нашей функции addCourse(int, float) пишется как Student::addCourse(int, float), так же как мое полное имя — Стефан Дэвис. Краткое имя этой функции — addCourse (int, float), а мое краткое имя —- Стефан. Имя класса в начале полного имени означает, что эта функция является членом класса Student (: : между именами функции и класса является просто символом-разделителем). Фамилия Дэвис после моего имени означает, что я являюсь членом семьи Дэвисов.

Существует и другое название полного имени — расширенное имя.

Мы можем определить функцию addCourse (int, float), которая не будет иметь ничего общего с классом Student; точно так же, как могут существовать люди с именем Стефан, которые не имеют ничего общего с моей семьей (это можно понимать и дословно: я знаю несколько Стефанов, которые не хотят иметь ничего общего с моей семьей).

Вы можете создать функцию с полным именем Teacher: :addCourse (int, float) или даже с именем Golf: :addCourse (int, float). Имя addCourse (int, float) без имени класса означает, что это обычная функция, которая не является членом какого-либо класса.

Вызов функций-членов

Прежде чем вызывать функции-члены класса, вспомните, как мы обращались к данным-членам классов:

```
class Student
{
public:
    int   semesterHours;
    float gpa;
};
Student s;
void fn(void)
{
    //обращение к данным-членам объекта s
    s.seinesterHours = 10;
    s.gpa = 3.0;
}
```

Обратите внимание, что наряду с именем переменной необходимо указать имя объекта. Другими словами, приведенный ниже фрагмент программы не имеет смысла.

```
Student s;
void fn(void)
{
        //этот пример ошибочен
    semesterHours = 10;
        // член какого объекта и какого класса?
    Student::semesterHours = 10;
        // теперь ясно, какого класса,
        // однако до сих пор не ясно,
        // какого объекта
}
```

Обращение к функциям-членам

Формально между данными-членами и функциями-членами нет никакого различия, что видно из приведенного ниже фрагмента.

```
Student s;
void fn(void)
{
    //все приведенные команды обращаются к объекту s
    s.semesterHours = 10;
    s.gpa = 3.0;
    s.addCourse(3, 4.0); // вызов функции-члена
}
```

Как видите, синтаксис вызова функции-члена такой же, как и синтаксис обращения к переменной-члену класса. Часть выражения, которая находится справа от точки, не отличается от вызова обычной функции. Единственное отличие — присутствие слева от точки имени объекта, которому принадлежит функция.

Факт вызова этой функции можно озвучить так: "s является объектом, на который действует addCourse () "; или, другими словами, объект s представляет собой студента, к записи которого добавляется новый курс. Вы не можете получить информацию о студенте или изменить ее, не указав, о каком конкретно студенте идет речь.

Вызов функции-члена без указания имени объекта имеет не больше смысла, чем обращение к данным-членам без указания объекта.

Доступ к членам из функции-члена

Я так и слышу, как вы повторяете про себя: "Нельзя обратиться к функции-члену без указания имени объекта! Нельзя обратиться к функции-члену без указания имени объекта! Нельзя..." Запомнив это, вы смотрите на тело функции-члена Student: :addCourse () и... что это? Ведь addCourse () обращается к членам класса, не уточняя имени объекта!

Возникает вопрос: все-таки можно или нельзя обратиться к члену класса, не указывая его обьекта? Уж поверьте мне, что нельзя. Просто, когда вы обращаетесь к члену класса Student из addCourse (), по умолчанию используется тот экземпляр класса, из которого вызвана функция addCourse (). Вы ничего не поняли? Вернемся к примеру:

```
#include "student.h"

float Student: :addCourse(int hours, float grade)
{
    float weightedGPA;
    weightedGPA = semesterHours * gpa;
    // добавим новый курс
```

```
    semesterHours += hours;
    weightedGPA += hours * grade;
    gpa = weightedGPA / semesterHours;
    return gpa;
}
int main(int argcs, char* pArgs[])
{
    Student s;
    Student t;

    s.addCourse(3, 4.0); // Поставим ему 4 балла
    t.addCourse(3, 2.5); // а этому дадим 2 с плюсом...
    return 0;
}
```

Когда addCourse () вызывается для объекта s, все сокращенные имена в теле этой функции считаются членами объекта s. Таким образом, обращение к переменной semesterHours внутри функции s.addCourse () в действительности является обращением к переменной s.semesterHours, а обращение к gpa — обращением к s.gpa. В следующей строке функции main (), когда addCourse () вызывается для объекта t того же класса Student, происходит обращение к членам класса t t.semesterHours и t.gpa.



Именование текущего объекта

Как функция-член определяет, какой объект является текущим? Это не магия и не шаманство — просто адрес этого объекта *всегда* передается функции-члену как *скрытый* первый аргумент. Другими словами, при вызове функции-члена происходит преобразование такого вида:

s.addCourse(3,2.5) равносильно Student::addCourse(&s, 3, 2.5)

(команда, приведенная в правой части выражения, синтаксически неверна; это просто изображение того, как компилятор видит выражение в левой части во внутреннем представлении).

Соответственно внутри функции, когда нужно узнать, какой именно объект является текущим, используется этот указатель. Тип текущего объекта — указатель на объект соответствующего класса.

Всякий раз, когда функция-член обращается к другому члену класса, не называя имени его объекта явно, компилятор считает, что данный член является членом *этого* (this) объекта. При желании вы можете явно обращаться к членам *этого* объекта, используя ключевое слово this. Так что функцию Student::addCourse() можно переписать следующим образом:

```
float Student::addCourse(int hours, float grade)
{
    float weightedGPA;
    weightedGPA = this->semesterHours * this->gpa;

    // добавим новый курс
    this->semesterHours += hours;
    weightedGPA += hours * grade;
    this->gpa = weightedGPA / this->semesterHours;
    return this->gpa;
}
```

Независимо от того, добавите ли вы оператор this-> в тело функции явно или нет, результат будет одинаков.



Объект, для которого вызывается функция-член, называется "текущим", и все имена членов, записанные в сокращенном виде внутри функции-члена, считаются членами текущего объекта. Другими словами, сокращенное обращение к членам класса интерпретируется как обращение к членам текущего объекта.

Разрешение области видимости

Символ `::` между именем класса и именем его члена называют *оператором разрешения области видимости,* поскольку он указывает, какой области видимости принадлежит член класса. Имя класса перед двоеточиями похоже на фамилию, тогда как название функции после двоеточия схоже с именем — такой порядок записи принят на востоке.

Оператор `::` можно использовать и для описания функции — не члена, использовав для этого пустое имя класса. В этом случае функция `addCourse()` должна быть описана как `::addCourse(int, float)`.

Обычно оператор `::` не обязателен, однако в некоторых ситуациях это не так. Рассмотрим следующий фрагмент кода:

```
// addCourse — перемножает количество часов и оценку
float addCourse(in hours, float grade)
{
    return hours*grade;
}
class Student
{
public:
    int    semesterHours;
    float gpa;

    // добавить пройденный курс к записи
    float addCourse(int hours, float grade)
    <
        // вызвать внешнюю функцию
        weightedGPA = addCourse(semesterHours, gpa);
        // вызвать ту же функцию для подсчета
        // оценки с учетом нового курса
        weightedGPA += addCourse(hours, grade);
        gpa = weightedGPA / semesterHours;

       // вернуть новую оценку
       return gpa;
    }
};
```

В этом фрагменте я хотел, чтобы функция-член `Student::addCourse()` вызывала функцию — не член `::addCourse()`. Без оператора `::` вызов функции `addCourse()` внутри класса `Student` приведет к вызову функции `Student::addCourse()`.



Функция-член может использовать для обращения к другому члену класса сокращенное имя, подразумевающее использование имени текущего экземпляра класса.

В данном случае вызов функции без указания имени класса приводит к тому, что она вызывает саму себя. Добавление оператора `::` в начале имени заставляет осуществить вызов глобальной версии этой функции (что нам и нужно):

```
// addCourse — перемножает количество часов и оценку
float addCourse(in hours, float grade)
{
    return hours*grade;
}
class Student
[
public:
    int   semesterHours;
    float gpa;

    // добавить пройденный курс к записи
    float addCourse(int hours, float grade)
    {
        // вызвать внешнюю функцию
        weightedGPA = ::addCourse(semesterHours, gpa);
        // вызвать ту же функцию для подсчета
        // оценки с учетом нового курса
        weightedGPA += ::addCourse(hours, grade);
        gpa = weightedGPA / semesterHours;

       // вернуть новую сценку
       return gpa;
    }
};
```

Это похоже на то, как если бы я звал Стефана в собственном доме. Все решили бы, что я зову самого себя: ведь в моем доме, естественно, подразумевается фамилия Дэвис. Если же я имею в виду какого-то другого Стефана, то должен сказать "Стефан Спупендайк" или "Стефан Мак-Суини" либо использовать какую-нибудь другую фамилию. Так же действует и оператор разрешения области видимости.



Расширенное имя функции включает в себя ее аргументы. Теперь же мы добавляем к полному имени еще и имя класса, к которому принадлежит функция.

Определение функции-члена

Функция-член может быть определена как внутри класса, так и отдельно от него. Когда функция определяется внутри класса, это выглядит так же, как и в файле student.h:

```
class Student
(
 public:
    int   semesterHours ;
    float gpa;

    // добавляем пройденный курс к записи
    float addCourse(int hours, float grade)
    {
        // подсчитываем суммарное время курсов
        // с учетом среднего балла
        float weightedGPA;
        weightedGPA = semesterHours * gpa;
```

```
        // добавляем новый курс
        semesterHours += hours;
        weightedGPA += grade * hours;
        gpa = weightedGPA / semesterHours;

        // вернуть новую оценку
        return gpa;
    }
};
```



Встраиваемые функции-члены

Функция-член, определенная непосредственно внутри класса, по умолчанию считается встраиваемой (подставляемой, inline) функцией (если только не оговорено обратное, например с помощью опций командной строки компилятора). Функции-члены по умолчанию считаются inline-функциями, потому что большинство функций-членов, определенных внутри класса, довольно малы, а такие маленькие функции являются главными кандидатами на подстановку.

Тело **inline-функции** подставляется компилятором непосредственно вместо оператора ее вызова. Подставляемая функция выполняется быстрее, поскольку от процессора не требуется осуществлять переход к телу функции. Однако при этом программы, использующие встроенные функции, занимают больше места, поскольку копии таких **inline-функций** определяются один-единственный раз, а подставляются вместо каждого вызова.

Есть еще одна техническая причина, по которой функции-члены класса лучше делать inline-функциями. Как вы помните, все структуры языка С обычно определяются в составе включаемых файлов с последующим использованием в исходных .с-файлах при необходимости. Такие включаемые файлы не должны содержать данных или тел функций, поскольку могут быть скомпилированы несколько раз. Использование же подставляемых функций во включаемых файлах вполне допустимо, поскольку их тела, как и макросы, подставляются вместо вызова в исходном файле. То же относится и к классам С++. Полагая **функции-члены**, определенные в описании классов, inline-функциями, мы избегаем описанной проблемы многократной компиляции.

Определение функций-членов вне класса

Для больших функций встраивание тела функции непосредственно в определение класса может привести к созданию очень больших и неудобочитаемых определений классов. Чтобы избежать этого, С++ предоставляет возможность определять тела функций-членов вне класса.

Реализовать тело функции вне класса Student можно, например, так:

```
class Student
{
public:
    int   semesterHours;
    float; gpa;
```

```
    // добавить пройденный курс к записи
    float addCourse(int hours, float grade)
};

// Добавляет информацию о пройденном курсе к
// полям класса Student
float Student::addCourse(int hours, float grade)
{
      float weightedGPA;
      weightedGPA = semesterHours * gpa;

      // добавить новый курс
      semesterHours += hours;
      weightedGPA += grade * hours;
      gpa = weightedGPA / semesterHours;
      return gpa;
}
```

Теперь объявление класса содержит только прототип функции addCourse(). При этом само тело функции находится в другом месте.

Объявление прототипа функции-члена по структуре не отличается от объявления прототипа любой другой функции, и, подобно всем объявлениям прототипов, обязательно.

В этом примере класс Student и функция Student::addCourse() определены в одном файле. Так можно делать, но такое расположение не очень распространено. Обычно класс Student определяется во включаемом файле, например Student.h, а тело функции может находиться в отдельном исходном файле, например Student.cpp.



Файл Student.cpp должен быть включен в состав вашего проекта вместе с другими файлами. Student.cpp будет скомпилирован в отдельный .obj-файл, который затем будет скомпонован с другими файлами в вашу программу на этапе сборки. Более детальное описание этого процесса можно найти в главе 6, "Создание функций".

Перегрузка функций-членов

Функции-члены могут перегружаться так же, как и обычные функции (обратитесь к главе 6, "Создание функций", если забыли, что это значит). Как вы помните, имя класса является частью полного имени, и все приведенные ниже функции вполне корректны.

```
class Student
{
public:
    //grade — возвращает текущую среднюю оценку
    float grade();
    //grade — устанавливает новое значение
    //оценки и возвращает предыдущее
    float grade(float newGPA);
    //... прочие члены-данные ...
};
class Slope
{
public:
    //grade — возвращает снижение оценки
```

```
    floatgrade() ;
    //...прочие члены-данные...
};

//grade — возвращает символьный эквивалент оценки
char grade(float value);

int main(int argcs, char* pArgs[])
{
    Student s;
    s.grade(3.5);         //Student::grade(float)
    float v = s . grade ();//Student::grade ()

    char c = grade(v);   //::grade (float)

    Slope o;
    float m = o.grade();//Slope::grade()
    return 0;
}
```

Полные имена вызываемых из main () функций указаны в комментариях.

Когда происходит вызов перегруженной функции, составляющими ее полного имени считаются не только аргументы функции, но и тип объекта, который вызывает функцию (если она вызывается объектом). Такой подход позволяет устранить неоднозначность при вызове функции.

В приведенном примере первые два вызова обращаются к функциям-членам Student::grade(float) И Student::grade() соответственно. Эти функции отличаются списками аргументов. Вызов функции s.grade() обращается к Student::grade(), поскольку тип объекта s — Student.

Третья вызываемая функция в данном примере— функция ::grade(float), не имеющая вызывающего объекта. Последний вызов осуществляется объектом типа Slope, и соответственно вызывается функция-член Slope::grade(float).

Глава 15

Создание указателей на объекты

В этой главе...

- ✓ Определение массивов указателей
- ✓ Объявление массивов объектов
- ✓ Объявление указателей на объекты
- ✓ Передача объектов функциям
- ✓ Возврат к куче
- ✓ Использование связанных списков
- ✓ Программа LinkedListData

Программисты на C++ все время создают массивы чего-либо. Формируются массивы целочисленных значений, массивы действительных значений; так почему бы не создать массив студентов? Студенты все время находятся в списках (причем гораздо чаще, чем им хотелось бы). Концепция объектов Student, стройными рядами ожидающих своей очереди, слишком привлекательна, чтобы можно было пройти мимо нее.

Определение массивов указателей

Массив является последовательностью идентичных объектов и очень похож на улицу с одинаковыми домами. Каждый элемент массива имеет индекс, который соответствует порядковому номеру элемента от начала массива. При этом первый элемент имеет нулевое смещение от начала массива, а значит, имеет индекс 0.

Массивы в C++ объявляются с помощью квадратных скобок, в которых содержится количество элементов в массиве.

```
int array[10];      //объявление массива из 10 элементов
```

К отдельному элементу массива можно обратиться, подсчитав количество домов от начала улицы до необходимого дома:

```
array[0] = 10;      //присвоить 10 первому элементу
array[9] = 20;      //присвоить 20 последнему элементу
```

В этом фрагменте первому элементу массива (элементу под номером 0) присваивается значение 10, а последнему — 20.



Не забывайте, что в C++ массив начинается с элемента с индексом 0 и заканчивается элементом, имеющим индекс, равный длине массива минус 1.

Если продолжить аналогию с домами, получится, что имя массива — это название улицы, а номер дома равнозначен номеру элемента в массиве. Таким же образом можно отождествить переменные с их адресом в памяти компьютера. Эти адреса могут быть определены и сохранены для последующего использования.

```
int variable;
                   //объявление целочисленной переменной
int* pVariable = &variable;
                   //сохранить ее адрес в pVariable
*pVariable = 10; // присвоить 10 целочисленной переменной,
                   // на которую указывает pVariable
```

Указатель pVariable был объявлен для того, чтобы хранить в нем адрес переменной variable. После этого целочисленной переменной, находящейся по адресу pVariable, присваивается значение 1С.

Использовав аналогию с домами в последний раз (честное слово, в последний!), мы получим:

- ✓ variable — это дом;
- ✓ pVariable — это листок с адресом дома;
- ✓ в последней строке примера происходит отправка сообщения, содержащего 10, по адресу, который находится на листке бумаги. Все почти так же, как на почте (единственное отличие состоит в том, что компьютер не ошибается адресом).

В главе 7, "Хранение последовательностей в массивах", описаны основы работы с массивами простых (встроенных) типов, а в главах 8, "Первое знакомство с указателями в C++", и 9, "Второе знакомство с указателями", подробно рассматриваются указатели.

Объявление массивов объектов

Массивы объектов работают так же, как и массивы простых переменных. В качестве примера можно использовать следующий фрагмент:

```
class Student
{
public:
    int   semesterHours;
    float gpa;
    float addCourse(int hours, float grade);
};
void someFn()
{
    //объявляем массив из 10 студентов
    Students[10];

    //Пятый студент получает 5.0 (повезло!)
    s[4].gpa = 5.0;

    //добавим еще один курс пятому студенту,
    //который на этот раз провалился...
    s[4].addCourse(3, 0.0) ;
}
```

В данном фрагменте s является массивом объектов типа Student. Запись s[4] означает пятый элемент массива, а значит, s[4] -gpa является усредненной оценкой пятого студента. В следующей строке с помощью функции s [4] . addCourse () пятому студенту добавляется еще один прослушанный и несданный курс.

Объявление указателей на объекты

Указатели на объекты работают так же, как и указатели на простые типы.

```
class Student
{
public:
    int   semesterHours;
    float gpa;
    float addCourse(int hours, float grade) {return 0.0};
};
int main(int argc, char* pArgs[])
{
    //создаем объект типа Student
    Student s;
    //создаем указатель на объект типа Student
    Student* pS;
    //заставляем pS указывать на наш объект s
    pS = &s;
    return 0;
}
```

Переменная pS является "указателем на объект типа `Student`"; другими словами, указателем `Student*`.

Разыменование указателей на объекты

По аналогии с указателями на простые переменные можно решить, что в приведенном ниже примере происходит обращение к усредненной оценке студента s.

```
int main(int argc, char* pArgs[])
{
                      //этот пример некорректен
    Student s;
    Student* pS= &s; //создаем указатель на объект s

    //обращаемся к члену gpa объекта, на который
    //указывает pS (этот фрагмент неверен)
    *pS.gpa = 3.5;

 return 0;
}
```

Как верно сказано в комментарии, этот код работать не будет. Проблема в том, что оператор ". " будет выполнен раньше оператора "*".

Для изменения порядка выполнения операторов в C++ используют скобки. Так, в приведенном ниже примере компилятор сначала выполнит сложение, а затем умножение.

```
int i = 2* (1+3);  //сложение выполняется до умножения
```

В применении к указателям скобки выполняют те же функции.

```
int main(int argc, char* pArgs[])
{
    Student s;
    Student* pS= &s; //создаем указатель на объект s

    //обращаемся к члену gpa того объекта, на который
    //указывает pS (теперь все работает правильно)
    (*pS).gpa = 3.5;
```

```
    return 0;
}
```

Теперь *pS вычисляет объект, на который указывает pS, а следовательно, .gpa обращается к члену этого объекта.

Использование стрелок

Использование для разыменования указателей на объекты оператора * со скобками будет прекрасно работать. Однако даже самые твердолобые программисты скажут вам, что такой синтаксис разыменования очень неудобен.

Для доступа к членам объекта C++ предоставляет более удобный оператор ->, позволяющий избежать неуклюжей конструкции со скобками и оператором *; таким образом, pS->gpa эквивалентно (*pS).gpa.

Этот оператор пользуется гораздо большей популярностью, поскольку его легче читать (хотя обе формы записи эквивалентны).

Передача объектов функциям

Передача указателей функциям — один из способов выразить себя в области указателей.

Вызов функции с передачей объекта по значению

Как вы знаете, по умолчанию C++ передает функции только значения аргументов. (Обратитесь к главе 6, "Создание функций", если вы этого не знали.) То же касается и составных, определенных пользователем объектов: они также передаются по значению.

```
#include "Student.h"

//передаем объект типа Student по значению
void someFn(Student valS)
{
    cout << "GPA = " << valS.GPA << "\n";
}

int main(int argc, char* pArgs[])
{
    Student s;
    s.semesterHours = 10;
    s.gpa = 3.0;

    //В следующей строке создается копия объекта s
    someFn(s);
    return 0;
}
```

В этом примере функция main() создает объект s, а затем передает его в функцию someFn().



Осуществляется передача по значению не самого объекта, а его копии.

Объект valS начинает свое существование внутри функции someFn() и является точной копией объекта s из main (). При этом любые изменения содержимого объекта valS никак не отражаются на объекте s из функции main {}.

Вызов функции с передачей указателя

Вместо того чтобы передавать объект по значению, можно передавать в функцию указатель на объект.

```
#include <stdio.h>
#include <iostream.h>

class Student
(
public:
    int   semesterHours;
    float gpa;
    float addCourse(int hours, float grade){return 0.0};
};
void someFn(Student* pS)
f
    pS->semesterHours = 10;
    p5->gpa = 3.0;
    pS->addCourse(3, 4.0); //вызываем функцию-член
}
int main(int argc, char* pArgs[])
{
    Student s;
    //передаем адрес объекта s функции someFn()
    someFn(&s);

    //передаем значение указателя pS
    Student* pS;
    pS = Ss;
    someFn(pS);
    return 0;
}
```

В этом примере аргумент, передаваемый в someFn (), имеет тип указателя на объект Student. Теперь вместо значения объекта s в функцию someFn () передается указатель на объект s. При этом соответственно изменяется и способ обращения к аргументам функции внутри ее тела: теперь для разыменования указателя ps используются операторы-стрелки.

Такой метод передачи аргументов схож с записью адреса дома s на листке бумаги и последующей передачей копии этого адреса в someFn ().

Зачем передавать указатель

Возможность передавать указатель на объект — это хорошо, но возникает вопрос: зачем это нужно? Оказывается, существует множество ситуаций, в которых предпочтительнее передавать указатель на объект вместо самого объекта, и некоторые из них будут описаны прямо сейчас.

Во-первых, передавая указатель на объект, функция получает возможность непосредственно менять содержимое аргумента. В противном случае может возникнуть ситуация, похожая на приведенную в этом примере:

```
#include <stdio.h>
#include <iostream.h>

class Student
{
public:
    int   semesterHours;
    float gpa;
    float addCourse(int hours, float grade){return 0.0;};
};

void someFn(Student copyS)
{
    copyS.semesterHours = 10;
    copyS.gpa           = 3.0;
    copyS.addCourse(3, 4.0);  //вызываем функцию-член
}
int main(int argc, char* pArgs[])
{
    Student s;
    s .gpa = 0.0;

    //отображаем значение s.gpa перед вызовом someFn
    cout « "Значение s.gpa = " « s.gpa << "\n";

    //передаем значение нашего объекта
    cout << "Вызываем функцию someFn(Student)\n";
    someFn(s);

    //значение s.gpa после вызова someFn остается равным 0
    cout << "Значение s.gpa = " << s.gpa << "\n";
    return 0;
}
```

В этом примере функции someFn () вместо адреса существующего объекта передается копия объекта s. Функция someFn () изменяет значение объекта, переданного ей. Однако, поскольку локальный объект copyS является всего лишь копией исходного объекта s, любые изменения этого объекта внутри someFn() никак не повлияют на s, а значит, не будут отражены в функции main О .

Результаты работы этой программы приведены ниже.

```
Значение s.gpa = 0
Вызываем функцию someFn(Student)
Значение s.gpa = 0
```

Переписав функцию someFn () так, чтобы ее аргументом являлся указатель на объект, мы решаем эту проблему:

```
#include <stdio.h>
#include <iostream.h>

class Student
{
 public:
    int   semesterHours;
    float gpa;
    float addCourse(int hours, float grade){return 0.0;};
};
```

```
void someFn(Student * pS)
{
    pS->semesterHours = 10;
    pS->gpa           = 3.0;
    pS->addCourse(3, 4.0);  //вызываем функцию-член
}

int main(int argc, char* pArgs[])
{
    Student s;
    s.gpa = 0.0;

    //отображаем значение s.gpa перед вызовом someFn
    cout << "Значение s.gpa = " << s.gpa << "\n";

    //передаем адрес нашего объекта
    cout << "Вызываем функцию someFn(Student)\n";
    someFn(&s);

   //теперь значение s.gpa после вызова someFn равно 3.0
   cout << "Значение s.gpa = " << s.gpa << "\n";
   return 0;
)
```

Теперь вместо создания еще одной копии функции someFn() передается адрес существующего объекта s (можно сказать, что вместо копии объекта функции передается копия адреса объекта).

Результат работы этой функции будет следующим:

```
Значение s.gpa = 0
Вызываем функцию someFn(Student)
Значение s.gpa =3.0
```

Передача объекта по ссылке

Оператор ссылки описан в главе 9, "Второе знакомство с указателями", и может применяться для пользовательских объектов так же, как и для всех остальных.

```
#include "Student.h"

//Все то же самое, но осуществляем передачу по ссылке
void someFn(Student& refS)
{
    refS.semesterHours = 10;
    refS.gpa = 3.0;
    refS.addCourse(3, 4.0);  //вызываем функцию-член
}
Student s;
int main(int argc, char* pArgs[])
{
    someFn(s);
    return 0;
}
```

В этой программе в функцию someFn() передается не копия объекта, а ссылка на него. Изменения, внесенные функцией someFn() в s, сохраняются внутри main().

Передача объекта по ссылке фактически передает в функцию адрес объекта s; при этом C++ самостоятельно разыменовывает переданный указатель.

Возврат к куче

Проблемы, возникающие при работе с указателями на простые переменные, распространяются и на указатели на объекты. В частности, необходимо гарантировать, что указатель ссылается на существующий корректный объект. Так, нельзя возвращать указатель на локально определенный объект, как это сделано в данном примере:

```
MyClass* myFunc()
{
    //эта функция не будет работать правильно
    MyClass mc;
    MyClass* pMC = &mc;
    return pMC;
}
```

После возврата из myFunc() объект mc выходит из области видимости, а значит, указатель, который возвращает myFunc(), указывает на несуществующий объект.

Использование кучи позволяет решить эту проблему:

```
MyClass*  myFunc()
{
    MyClass* pMC = new MyClass;
    return pMC;
}
```



С помощью кучи можно выделять память для объектов в самых разнообразных ситуациях.

Использование связанных списков

Связанный список является второй по распространенности структурой после массива. Основным преимуществом связанного списка служит отсутствие необходимости задавать фиксированный размер на этапе компиляции: связанный список может уменьшаться и увеличиваться в зависимости от потребностей программы. Цена такой гибкости — сложность обращения, поскольку использовать элементы связанного списка гораздо сложнее, чем элементы массива.

Массив

Массив очень удобен для хранения набора объектов, так как позволяет просто и быстро обратиться к отдельным его элементам:

```
MyClass mc[100];     //выделяем место под 100 элементов
mc[n];               //обратиться к (n+1)-му элементу списка
```

Однако у массивов есть много весомых недостатков.

Основным недостатком массивов является фиксированность их длины. Можно спрогнозировать количество элементов в массиве, необходимое для работы программы, однако, будучи однажды определенным, это количество не может быть изменено.

```
void fn(int nSize)
{
    //создаем массив из n объектов
    //MyClass
    MyClass* pMC = new MyClass[n];
    //теперь длина массива не может
```

```
    //быть изменена
}
```

Кроме того, все элементы массива должны быть одного типа. Невозможно хранить объекты типа MyClass и YourClass в одном массиве.

И наконец, крайне трудно вставить объект в середину массива. Для добавления или удаления объекта программа должна перенести соседние элементы вперед или назад, чтобы создать или уничтожить пустое место в середине массива. (Представьте себе процесс вставки дома внутрь застроенного квартала, и вы поймете, о чем я говорю.)

Однако существует и альтернативная структура данных, лишенная указанных недостатков. Конструкция, которая может заменить массив, называется связанным списком.

Связанный список

Дети, держащие друг друга за руки и переходящие дорогу, используют тот же принцип, что и связанный список. В связанном списке каждый объект связан со следующим объектом в цепочке. Учитель в этом случае представляет собой головной элемент, указывающий на первый элемент списка.

Не всякий класс может быть использован для создания связанного списка. Связываемый класс объявляется так, как показано в приведенном ниже фрагменте.

```
class LinkableClass
{
    public:
        LinkableClass* pNext;
        //прочие члены класса
};
```

Ключевым в этом классе является указатель на объект класса LinkableClass. На первый взгляд несколько необычно выглядит то, что класс содержит указатель сам на себя. В действительности в этом объявлении подразумевается, что каждый объект класса содержит указатель на другой объект этого же класса.

Указатель pNext и есть тот элемент, с помощью которого дети объединяются в цепочки. Фигурально выражаясь, можно сказать, что список детей состоит из некоторого количества объектов, каждый из которых имеет тип "ребенок". Каждый ребенок указывает на следующего ребенка.

Головной указатель является указателем типа LinkableClass*, и если продолжать аналогию с цепочкой детей, то можно сказать, что учитель указывает на объект класса "ребенок" (любопытно отметить, что сам учитель не является ребенком — головной указатель не обязательно должен иметь тип LinkableClass).



Не забывайте инициализировать указатели значением 0. Указатель, содержащий ноль, называется "пустым". Обычно попытка обращения по адресу 0 вызывает аварийную остановку программы.



Преобразование целочисленного нуля в тип LinkableClass* не обязательно. C++ воспринимает 0 как значение любого типа (в частности, как "универсальный указатель").

Приведенная ниже простая функция, добавляющая переданный ей аргумент в начало списка, поможет вам разобраться со связанными списками.

```
void addHead(LinkableClass* pLC)
{
    pLC->pNext = pHead
    pHead = pLC;
}
```

Другие операции над связанным списком

Добавление объекта в начало списка является самой простой операцией со связанным списком. Гораздо сложнее добавить элемент в конец списка:

```
void addTail(LinkableClass* pLC)
{
    //указатель на начало списка
    LinkableClass* pCurxent = pHead;
    //будем просматривать список, пока не найдем
    //последний элемент списка — он должен содержать
    //0 в указателе pNext
    while (pCurrent->pNext != fLinkableClass*) 0]
    {
        pCurrent = pCurrent->pNext;
    }
    //теперь заставим этот объект указывать на LC
    pCurrent->pNext = pLC;
    //присвоим 0 указателю на следующий объект
    //в новом конечном элементе, пометив его
    //как последний в списке
    pLC->pNext = (LinkableClass&*)0;
}
```

Тело функции addTail () начинается с просмотра всего списка с самого начала до элемента, значение поля pNext которого равно нулю. Этот элемент и является последним в списке. После этого функция addTail () добавляет объект pLC* к концу списка.

(В действительности приведенная функция содержит одну ошибку. Кроме проверки элементов списка на равенство нулю, необходимо проверять и головной элемент списка — pHead. Ноль в pHead означает, что список пуст.)

Функция remove () схожа с предыдущей. Она удаляет указанный объект из списка и возвращает 1, если удаление прошло без ошибок, и 0 в противном случае.

```
void remove(LinkableClass* pLC)
{
    LinkableClass* pCurrent = pHead;
    //если список пуст, мы не найдем в нем *pLC
    if (pCurrent == (LinkableClass*) 0)
    {
        return 0;
    }
    //просмотрим весь список до конца в поисках
    //указанного элемента
    while (pCurrent->pNext);
    {
        //если следующий объект является искомым объектом
        if(pLC==pCurrent->pNext)
        {
            //...то заставим текущий объект указывать
            //на следующий за ним
            pCurrent->pNext = pLC->pNext;
            //это не обязательно, но лучше удалить указатель
            //на следующий объект в удаляемом элементе
            pLC->pNext = (LinkableClass&*)0;
            return 1;
        }
    }
    return 0;
}
```

Сначала функция remove() проверяет, не пуст ли список. Если список пуст, функция возвращает 0 (поскольку в этом случае объекта *pLC точно нет в списке). Если список не пуст, remove O просматривает все элементы списка, пока не найдет искомый объект. Если функция находит объект, она присваивает текущему указателю pNext значение указателя pNext из найденного объекта. Было бы неплохо, если бы, внимательно рассмотрев приведенный фрагмент, вы могли сказать, в каком случае он будет работать неправильно.

Свойства связанных списков

Связанные списки имеют все те преимущества, которых нет у массивов. Связанные списки могут удлиняться и укорачиваться по мере добавления или удаления элементов. Добавление объекта в середину списка выполняется легко и быстро — при этом нет необходимости перемещать существующие элементы. Кроме того, сортировка элементов в связанном списке проводится гораздо быстрее, чем в массиве элементов.

Недостаток работы со связанными списками состоит в том, что поиск определенного элемента затруднен по сравнению с аналогичной операцией в массивах. К элементам массива легко обратиться посредством индекса, тогда как в связанном списке это сделать невозможно. Иногда программа должна прочитать весь список для поиска одного элемента.

Программа LinkedListData

Программа LinkedListData использует связанный список для хранения списка объектов, содержащих имена студентов и их номера социального страхования.

```
// LinkedListData — хранит имена студентов
//                  в связанном списке объектов
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet — содержит имена и номера социального
//               страхования
class NameDataSet
{
   public:
      char szFirstName[128];
      char szLastName [128];
      int  nSocialSecurity;

      // связь со следующим элементом списка
      NameDataSet* pNext;
};

// указатель на первый элемент списка
NameDataSet* pHead = 0;

// addTail — добавляет новый элемент в связанный список
void addTail(NameDataSet* pNDS)
{
    // сразу же обнулим указатель на следующий элемент,
    // поскольку этот элемент будет последним в списке
    pNDS->pNext = 0;

    // если список пуст,
    // то головной указатель должен будет
```

```
    // указывать на вставляемый элемент
    if (pHead == 0)
    {
        pHead = pNDS;
        return;
    }
    // если список не пуст, найдем последний элемент списка
    NameDataSet* pCurrent = pHead;
    while(pCurrent->pNext)
    {
        pCurrent = pCurrent->pNext;
    }
    // добавим текущий элемент к концу списка
    pCurrent->pNext = pNDS;
}

// getData — считывает имя и номер социального
//           страхования; возвращает ноль,
//           если больше нечего считывать
NameDataSet* getData()
{
    // создаем новый объект для заполнения
    NameDataSet* pNDS = new NameDataSet;

    // считаем имя
    cout << "\nВведите имя: ";
    cin  >> pNDS->szFirstName;

    // если в поле имени введено 'exit'...
    if ((stricmp(pNDS->szFirstName, "exit") == 0))
    {
        // ...уничтожить еще пустой объект...
        delete pNDS;

        // ...и вернуть ноль, чтобы прервать цикл ввода
        return 0;
    }

    // считаем оставшиеся члены
    cout << "Введите фамилию: ";
    cin  >> pNDS->szLastName;

    cout << "Введите номер социального страхования: ";
    cin  >> pNDS->nSocialSecurity;

    // обнулим указатель на следующий элемент
    pNDS->pNext = 0;

    // вернем адрес созданного объекта
    return pNDS;
}

// displayData - выводим один экземпляр класса NameDataSet
void displayData(NameDataSet* pNDS)
{
    cout; << pNDS->szFirstName
         << " "
         << pNDS->szLastName
```

```
        << "/"
        << pNDS->nSocialSecurity
        «  "\n";
}

int main(int argc, char* pArgs[])
{
    cout << "Считываем имя/номер социального страхования \п"
         « "Введите 'exit' в поле имени для выхода\n";

    // создадим объект класса NameDataSet
    NameDataSet* pNDS;
    while (pNDS = getDataO)
    f
        // добавим его в конец списка объектов
        // NameDataSet
        addTail(pNDS);
    }

    // поочередно выведем все элементы списка
    // (выполнение прекратится, когда указатель
    // на следующий элемент будет равен нулю)
    cout « "Элементы:\п";
    pNDS = pHead;
    while(pNDS)
    {
        // отобразим текущий элемент
        displayData(pNDS);

        // получим адрес следующего элемента
        pNDS = pNDS->pNext;
    }
    return 0;
}
```

Несмотря на внушительную длину, программа LinkedListData относительно проста. Функция main () начинается с вызова функции getData (), которая считывает элемент NameDataSet с клавиатуры. Если пользователь вводит строку "exit" в поле имени, getData () возвращает ноль. Функция main () вызывает функцию addTail (), чтобы добавить элемент, который вернула getData (), в конец связанного списка.

Если от пользователя больше не поступает элементов NameDataSet, функция main () выводит на экран все элементы списка, используя функцию displayData ().

Функция getData() выделяет из кучи пустой объект класса NameDataSet. После этого getData () ожидает ввода имени для записи его в соответствующее поле нового объекта. Если пользователь вводит в поле имени строку "exit", функция уничтожает последний созданный объект и возвращает 0. В противном случае getData() считывает фамилию и номер социального страхования, после чего обнуляет указатель pNext и передает управление вызывающей функции.



Никогда не оставляйте связывающие указатели не проинициализированными! Старая поговорка программистов гласит: "Не уверен — обнули".

Функция addTail () из приведенного примера схожа с предыдущей версией addTail O, однако в отличие от нее она проверяет, не является ли данный список пустым. Если указатель pHead нулевой, то addTail O записывает в него указатель на текущий элемент и прекращает выполнение.

Функция displayData () неоднократно встречалась нам ранее.

Глава 16

Защищенные члены класса: не беспокоить!

В этой главе...

- ✓ Защищенные члены
- ✓ Чем хороши защищенные члены
- ✓ Обращение к защищенным членам

В главе 13, "Классы в C++", рассматривалась концепция классов. Ключевое слово public было описано как часть объявления класса, т.е. просто как то, что следует делать. В этой главе вы узнаете, что ключевому слову public есть альтернативы.

Защищенные члены

Члены класса могут быть помечены как защищенные, что делает их недоступными извне класса. В отличие от защищенных, открытые (public) члены класса доступны для всех.

Зачем нужны защищенные члены

Для того чтобы понять смысл защиты членов класса, нужно вспомнить, каковы цели объектно-ориентированного программирования.

- ✓ Зашита внутренних элементов класса от внешних функций. Ведь когда вы проектируете микроволновую печь (или что-нибудь другое), то оснащаете ее по возможности простым интерфейсом с внешним миром и прячете содержимое в металлический ящик. Так делается для того, чтобы другие не могли поломать микроволновую печь. Защита членов класса выполняет роль железного ящика.
- ✓ Создание класса, способного полноценно управлять своими внутренними членами. Несколько непоследовательно требовать от класса полноценной работы и ответственности за ее результаты и одновременно позволять внешним функциям манипулировать его внутренними членами (это то же самое, что и требовать от создателя микроволновой печи нести ответственность за мои непрофессиональные манипуляции с элементами ее внутреннего устройства).
- ✓ Сокращение до минимума внешнего интерфейса класса. Гораздо проще изучать и использовать класс, который имеет ограниченный интерфейс (а интерфейсом класса являются его открытые члены). Защищенные члены скрыты от пользователя, и их не надо помнить (в противном случае интерфейсом становится весь класс). Такой подход называется *абстракцией*, которая описана в главе 8, "Первое знакомство с указателями в C++".

✓ Уменьшение уровня взаимосвязи между классом и внешней программой. Ограничив взаимосвязь класса с внешним кодом, при необходимости гораздо проще заменить класс каким-либо другим.

Я так и слышу, как поклонники функционального подхода говорят: "Не нужно делать ничего противоестественного! Достаточно потребовать от программиста, чтобы он попросту не трогал некоторые члены класса".

Все это верно в теории, однако не оправдывается на практике. Программисты начинают писать программы, будучи переполненными благих намерений, однако приближение сроков сдачи программы заставляет их все больше разочаровываться в отсутствии возможности прямого доступа к защищенным членам класса.

Как устроены защищенные члены

Добавление в класс ключевого слова public делает все находящиеся за ним члены класса открытыми, а значит, доступными для функций — не членов класса. Использовав ключевое слово protected, вы делаете все последующие члены класса защищенными, т.е. недоступными для функций, которые не являются членами класса. Переключаться между защищенными и открытыми членами класса можно сколько угодно.

Допустим, у нас есть класс student. В приведенном ниже примере представлены все необходимые возможности, которые нужны классу, описывающему студента (за исключением разве что функций spendMoney (тратить деньги) и drinkBeer (пить пиво), которые тоже являются свойствами студента):

addCourse(int hours, float grade) — добавить пройденный курс;
grade() — вернуть текущую среднюю оценку;
hours() — вернуть количество пройденных часов.

Оставшиеся члены класса Student можно объявить как защищенные, чтобы другие функции не могли "лезть" во внутренние дела класса Student.

```
class Student
{
public:
    // grade — возвращает текущую среднюю оценку
    float grade()
    {
        return gpa;
    }
    // hours — возвращает количество прослушанных часов
    int hours()
    {
        return semesterHours;
    }
    // addCourse — добавляет к записи студента
    // прослушанный курс
    float addCourse(int hours, float grade)
    //приведенные ниже члены недоступны для внешних функций
protected:
    int semesterHours;  //количество прослушанных часов
    float gpa;          //средняя оценка
};
```

Теперь члены semesterHours и gpa доступны только из других членов класса Student, и приведенный ниже пример работать не будет.

```
Student s;
int main(int argc, char* pArgs[])
{
```

```
    //повысим свой рейтинг (но не слишком сильно,
    //иначе никто не поверит)
    s.gpa = 3.5;          //вызовет ошибку при компиляции
    float gpa = s.grade(); //эта открытая функция
        // считывает значение переменной, но вы не можете
        // непосредственно изменить ее значение извне
    return 0;
}
```

При попытке этой программы изменить значение gpa на этапе компиляции будет выдано сообщение об ошибке.



Считается признаком хорошего тона не полагаться на значение защиты по умолчанию, а определить в самом начале объявления класса ключевое слово public или private. Обычно класс начинают описывать с открытых членов, формируя интерфейс класса. Описание защищенных членов класса выполняется позже.



Члены класса могут быть защищены извне и с помощью еще одного ключевого слова — private. Кстати, по умолчанию при описании класса его члены считаются описанными именно как private. Разница между protected и private станет ясной при изучении наследования в главе 21, "Наследование классов".

Чем хороши защищенные члены

Теперь, когда вы немного познакомились с защищенными членами, я приведу аргументы, обосновывающие их использование.

Защита внутреннего устройства класса

Ключевое слово protected позволяет исключить возможность установки gpa равным не допустимому для этой величины значению. Внешнее приложение сможет добавить курс, но не сможет изменить значение среднего балла непосредственно. Если имеется необходимость непосредственного изменения значения gpa, класс может предоставить открытую функцию, предназначенную для этой цели, например:

```
class Student
{
public:
    // grade — делает то же, что и раньше
    float grade()
    {
        return gpa;
    }
    //даем возможность изменения средней оценки
    float grade(float newGPA)
    {
        float oldGPA = gpa;
        //Проверяем допустимость значения
        if (newGPA > 0 && newGPA <= 5.0)
        {
            gpa = newGPA;
        }
        return oldGPA;
    }
```

```
    //... все остальное остается без изменений
    protected:
    int semesterHours;  //количество прослушанных часов
    float gpa;
};
```

Добавление новой функции grade(float) позволяет внешним приложениям изменять содержимое gpa. Заметьте, что класс все равно не позволяет внешним функциям полностью контролировать содержимое своих защищенных членов. Внешнее приложение не может присвоить gpa любое значение, а только то, которое лежит в диапазоне между 0 и 5.0.

Теперь класс Student обеспечивает внешний доступ к своим внутренним членам, одновременно не позволяя присвоить им недопустимое значение.

Классы с ограниченным интерфейсом

Теперь наш класс предоставляет *ограниченный* интерфейс. Чтобы использовать класс, достаточно знать, каковы его открытые члены, что они делают и какие аргументы принимают. Это значительно уменьшает количество информации, которую необходимо помнить для работы с классом.

Кроме того, иногда изменяются условия работы программы либо выявляются новые ошибки, и программист должен изменить содержимое членов класса (если не логику его работы). В этом случае изменение только защищенных членов класса не вызывает изменений в коде внешнего приложения.

Обращение к защищенным членам

Может случиться так, что потребуется предоставить некоторым внешним функциям возможность обращения к защищенным членам класса. Для такого доступа можно воспользоваться ключевым словом friend (друг).

"Друг всегда уступить готов место в шлюпке и круг.,."

Иногда внешним функциям требуется прямой доступ к данным-членам. Без некоторого механизма "дружественности" программист был бы вынужден объявлять такие члены открытыми для всех, а значит, обращаться к этим членам могла бы любая внешняя функция.

Это похоже на то, как вы порой оставляете соседям ключ от своего дома на время отпуска, чтобы они иногда проверяли его. Давать ключи не членам семьи не совсем хорошо, однако это куда лучше, чем оставлять дом открытым.

Объявление друзей должно находиться в классе, который содержит защищенные члены. Такое объявление выполняется почти так же, как и объявление обычных прототипов, и должно содержать расширенное имя друга, включающее типы аргументов и возвращаемого значения. В приведенном ниже примере функция initialize() получает доступ ко всем членам класса Student.

```
class Student;
{
    friend void initialize(Student*);
public:
   //те же открытые члены, что и раньше
protected:
    int semesterHours; //количество часов в семестре
    float gpa;
};
```

```
//эта функция — друг класса Student
//и имеет доступ к его защищенным членам
void initialize(Student *pS)
{
    pS->gpa = 0;            //теперь эти строки законны
    pS->semesterHours = 0;
}
```

Одна и та же функция может одновременно быть объявлена другом нескольких классов. Это может быть удобно, например, для связи двух классов. Правда, такого рода связь не очень приветствуется, поскольку делает оба класса зависимыми друг от друга. Однако, если два класса взаимосвязаны по своей природе, их объединение может оказаться не столь плохим решением.

```
class Student;
class Teacher
{
    friend void registration();
protected:
    int noStudents;
    Student *pList[100];
public:
    void assignGrades();
};
class Student
{
    friend void registration();
public:
    //те же открытые члены, что и раньше
protected:
    Teacher *pT;
    int semesterHours; //количество часов в семестре
    float gpa;
};
```

В данном примере функция registration() может обращаться к обоим классам — и student и Teacher, связывая их на этапе регистрации, но при этом не входя в состав этих классов.

Обратите внимание, что в первой строке примера объявляется класс Student, но не объявляются его члены. Запомните: такое описание класса называется предварительным и в нем описывается только имя класса. Предварительное описание нужно для того, чтобы другие классы, такие, например, как Teacher, могли обращаться к классу Student. Предварительные описания используются тогда, когда два класса должны обращаться один к другому. Функция-член одного класса может быть объявлена как друг некоторого другого класса следующим образом:

```
class Teacher
{
//те же члены, что и раньше
public:
    void assignGrades();
};
class Student
{
    friend void Teacher::assignGrades();
public:
    //те же открытые члены, что и раньше
protected:
    int semesterHours; //количество часов в семестре
```

```
    float gpa;
};
void Teacher::assignGrades();
{
    //эта функция имеет доступ к
    //защищенным членам класса Student
}
```

В отличие от примера с функциями — не членами, функция-член класса должна быть объявлена перед тем, как класс Student объявит ее другом.

Существующий класс может быть объявлен как друг некоторого другого класса целиком. Это означает, что все функции-члены класса становятся друзьями другого класса, например:

```
class Student;
class Teacher
{
protected:
    int  noStudents;
    Student  *pList[100];
public:
    void assignGrades();
};
class Student
{
    friend class Teacher;
public:
    //те же открытые члены, что и раньше
protected:
    Teacher *pT;
    int semesterHours; //количество часов в семестре
    float gpa;
};
```

Теперь любая функция-член класса Teacher имеет доступ ко всем защищенным членам класса Student. Объявление одного класса другом другого неразрывно связывает два класса.

Глава 17

Создание и удаление объектов: конструктор и деструктор

В этой главе...

- ✓ Создание объектов
- ✓ Использование конструкторов
- ✓ Что такое деструктор

Объекты в программе создаются и уничтожаются так же, как и объекты реального мира. Если класс сам отвечает за свое существование, он должен обладать возможностью управления процессом уничтожения и создания объектов. Программистам на C++ повезло, поскольку C++ предоставляет необходимый для этого механизм (хотя, скорее всего, это не удача, а результат разумного планирования языка). Прежде чем начинать создавать и уничтожать объекты в программе, обсудим, что значит "создавать объекты".

Создание объектов

Некоторые подчас теряются в терминах *класс* и *объект.* В чем разница между этими терминами? Как они связаны?

Я могу создать класс Dog, который будет описывать соответствующие свойства лучшего друга человека. К примеру, у меня есть две собаки. Это значит, что мой класс Dog содержит два экземпляра — Труди и Скутер (надеюсь, что два: Скутера я не видел уже несколько дней...).



Класс описывает тип предмета, а *объект* является экземпляром класса. Dog является классом, а Труди и Скутер — объектами. Каждая собака представляет собой отдельный объект, но существует только один класс Dog, при этом не имеет значения, сколько у меня собак.

Объекты могут создаваться и уничтожаться, а классы попросту существуют. Мои собаки Труди и Скутер приходят и уходят, а класс Dog (оставим эволюцию в стороне) вечен.

Различные типы объектов создаются в разное время. Когда программа начинает выполняться, создаются глобальные объекты. Локальные объекты создаются, когда программа сталкивается с их объявлением.



Глобальный объект является объектом, объявленным вне каких-либо функций. Локальный объект объявляется внутри функции, а следовательно, является локальным для функции. В приведенном ниже примере переменная me является глобальной, а переменная noMe — локальной по отношению к pickOne().

```
int  me;
void pickOne ()
{
    int noMe;
}
```



Согласно правилам С глобальные объекты по умолчанию инициализируются нулевыми значениями- Локальные объекты, т.е. объекты, объявленные внутри функций, не имеют инициализирующих значений. Такой подход, вообще говоря, для классов неприемлем.

С++ позволяет определить внутри класса специальную функцию-член, которая автоматически вызывается при создании объекта этого класса. Эта функция-член называется конструктором и инициализирует объект, приводя его в некоторое необходимое начальное состояние. Кроме конструктора, в классе можно определить деструктор, который будет вызываться при уничтожении объекта. Эти две функции и являются предметом обсуждения данной главы.

Использование конструкторов

Конструктор — это функция-член, которая вызывается автоматически во время создания объекта соответствующего класса. Основной задачей конструктора является инициализация объекта, приводящая его в некоторое корректное начальное состояние.

Зачем нужны конструкторы

Объект можно проинициализировать на этапе его объявления, как сделал бы программист на С:

```
struct Student
{
    int   semesterHours;
    float gpa;
};
void fn()
{
    Student s = {0,0};
    // ...продолжение функции...
}
```

Этот фрагмент кода не будет работать для настоящего класса С++, поскольку внешнее приложение не имеет доступа к защищенным членам класса. Приведенный ниже фрагмент некорректен.

```
class Student
{
public:
    // ...открытые члены...
protected:
    int   semesterHours;
    float gpa;
};
void fn()
{
    Student s = {0,0};      // неправильно, так как
                            // данные-члены недоступны
    // ...продолжение функции...
}
```

В данном случае функция — не член `fn()` не имеет права обращаться к членам `semesterHours` и `gpa`, поскольку они являются защищенными (а функция не объявлена в качестве друга класса).

Однако, как уже отмечалось, вы можете обеспечить класс инициализирующей функцией, которая будет вызываться в процессе создания объекта. Поскольку эта функция является членом класса, она имеет доступ к защищенным членам. Такой способ инициализации продемонстрирован в приведенном ниже примере.

```
class Student
{
public:
    void init()
    {
        semesterHours = 0;
        gpa = 0.0;
    }
    // ...остальные открытые члены...
protected:
    int  semesterHours;
    float gpa;
};
void fn()
{
    Student s;    // создаем объект ...
    s.init{);     // ... и инициализируем его
    // ...продолжение функции...
}
```

Однако при таком подходе ответственность за инициализацию объекта снимается с класса. Другими словами, класс должен полагаться на то, что внешнее приложение вызовет функцию-член init (). В противном случае объект останется не проинициализированным и будет содержать непредсказуемые значения.

Для того чтобы избежать этой неприятности, ответственность за вызов инициализирующей объект функции необходимо переложить с приложения на компилятор. Всякий раз при создании объекта компилятор может вставлять в код специальную инициализирующую функцию — а это и есть конструктор!

Работа с конструкторами

Конструктор ~ это специальная функция-член, которая автоматически вызывается во время создания объекта. Конструктор должен иметь то же имя, что и класс. (Таким образом компилятор сможет определить, что именно эта функция-член является конструктором. Конечно, создатели C++ могли сформулировать это правило как угодно, например, так: "Конструктором является функция с именем init()". Как именно определено правило, не имеет значения; главное — чтобы конструктор мог быть распознан компилятором.) Еще одним свойством конструктора является то, что он не возвращает никакого значения, поскольку вызывается автоматически (если бы конструктор и возвращал значение, его все равно некуда было бы записать).

Класс с использованием конструктора продемонстрирован в следующем примере:

```
#include <iostream.h>
class Student
!
public:
    Student()
    {
        cout << "Конструируем объект Student\n";
        semesterHours = 0;
        gpa = 0.0;
    }
```

```
    // ...остальные открытые члены...
protected:
    int   semesterHours;
    float gpa;
};
void fn()
{
    Student s;    // создаем и инициализируем объект
    // ...продолжение функции...
}
```

В этом примере компилятор сам вызывает конструктор `Student::Student(]` в том месте, где объявляется объект s.

Этот простой конструктор реализован в виде встроенной (inline) функции. Конструктор можно создать и как обычную функцию с телом, вынесенным из объявления класса:

```
#include <iostream.h>
class Student
{
public:
    Student();
    // ...остальные открытые члены...
protected:
    int   semesterHours;
    float gpa;
i;
Student::StudentO
(
    cout << "Конструируем Student\n";
    semesterHours = 0;
    gpa = 0.0;
}
voidfn ()
{
    Student s;    // создаем и инициализируем объект
    // ...продолжение функции...
}
int main(int argcs, char* pArgs[])
{
    fn();
    return 0;
}
```

В данном примере добавлена небольшая функция `main()`, чтобы эту тестовую программу можно было запустить. Настоятельно рекомендую пройти эту программу в пошаговом режиме отладчика перед тем, как двигаться дальше.

Инструкции по работе с отладчиком GNU C++ приведены в главе 29, "Десять способов избежать ошибок".

Выполняя этот пример в пошаговом режиме, дойдите до строки с объявлением объекта s. Выполните команду отладчика Шаг с заходом в функции (Step into), и управление как по волшебству перейдет к функции `Student::Student()` (если у вас inline-версия конструктора, убедитесь, что при компиляции включена опция Outline inline functions, с помощью которой тело функции выносится из объявления класса; в противном случае код конструктора будет подставлен в тело программы и вы не сможете попасть в конструктор). Продолжайте выполнение конструктора в пошаговом режиме. Когда функция закончится, управление перейдет к следующей за объявлением объекта класса строке.

В одной строке может быть объявлено несколько объектов. Выполните, например, о пошаговом режиме следующее:

```
void f.n()
{
    Student s[5];    //создаем массив объектов
   //...продолжение функции...
}
```

Вы увидите пять вызовов конструктора — по одному для каждого элемента массива.



В приведенной далее программе используются команды вывода сообщений, информирующие о вызове конструктора, на случай, если вы не можете (или не хотите) запустить отладчик. Эффект будет хотя и не столь впечатляющим, но от этого не менее убедительным.

Конструктор может быть запущен только автоматически! Нельзя вызывать конструктор как обычную функцию-член, а значит, и повторно инициализировать объект класса Student:

```
void fn()
{
    Student s;      // создаем и инициализируем объект
                    // ...продолжение функции...
    s.Student();    // пытаемся реинициализировать его,
                    // но компилятор сообщит об ошибке
}
```

Объявление конструктора не должно содержать тип возвращаемого значения — и даже тип void.

Если класс имеет данные-члены, которые являются объектами другого класса, конструкторы для этих объектов также будут вызваны автоматически. Обратите внимание на следующий пример, в который добавлены команды вывода сообщений, позволяющие увидеть, в каком порядке создаются объекты:

```
#induce <iostream.h>
class Student
{
public:
    Student()
    {
        cout << "Конструируем Student\n";
        semesterHours = 0;
        gpa = 0.0;
    )
    // ...остальные открытые члены...
protected:
    int   semesterHours;
    float gpa;
};
class Teacher
{
public:
    Teacher()
    {
        cout << "Конструируем Teacher\n";
    }
};
class TutorPair
```

```
{
public:
    TutorPair()
    {
        cout << "Конструируем TutorPair\n";
        noMeetings = 0;
    }
protected:
    Student student;
    Teacher teacher;
    int noMeetings;
};
int main(int argc, char* pArgs[])
{
    TutorPair tp;
    cout << "Возвращаемся в main()\n";
    return 0;
}
```

В результате работы этой программы на экран будут выведены следующие сообщения:

```
Конструируем Student
Конструируем Teacher
Конструируем TutorPair
Возвращаемся в main()
```

Создание объекта tp в main;) вызывает конструктор TutorPair автоматически. Перед тем как управление будет передано телу конструктора TutorPair, вызываются конструкторы для объектов-членов student и teacher.

Конструктор student вызывается первым, поскольку объект этого класса объявлен первым. Затем вызывается конструктор Teacher. И только после создания этих объектов управление переходит к конструктору класса TutorPair, который теперь может конструировать оставшуюся часть объекта.



Это не означает, что TutorPair отвечает за инициализацию объектов student и teacher. Каждый класс отвечает за инициализацию своего объекта, где бы тот не создавался.

Что такое деструктор

Объекты класса уничтожаются так же, как и создаются. Если класс может иметь конструктор для выполнения начальных установок, то он может содержать и специальную функцию для уничтожения объекта. Такая функция-член называется деструктором.

Зачем нужен деструктор

Класс может затребовать для своего объекта некоторые ресурсы с помощью конструктора; эти ресурсы должны быть освобождены при уничтожении объекта. Например, если конструктор открывает файл, перед окончанием работы с объектом класса или программы этот файл следует закрыть. Возможен и другой вариант: если конструктор берет память из кучи, то она должна быть освобождена перед тем, как объект перестанет существовать. Деструктор позволяет делать это автоматически, не полагаясь на вызов необходимых функций-членов в программе.

Работа с деструкторами

Деструктор имеет то же имя, что и класс, но только с предшествующим ему символом тильды (~) (С++ последователен и здесь: ведь символ тильды не что иное, как символ оператора "нет", т.е. деструктор — это отрицание конструктора). Как и конструктор, деструктор не имеет типа возвращаемого значения. С учетом сказанного деструктор класса Student будет выглядеть так:

```
class Student
{
public:
    Stucent()
    {
        semesterHours = 0;
        gpa = 0.0;
    }
    ~Student()
    {
        //любые используемые ресурсы освобождаются здесь
    }
    // ...остальные открытые члены...
 protected:
    int   semesterHours;
    float gpa;
};
```

Деструктор вызывается автоматически, когда объект уничтожается или, если говорить языком С++, происходит его деструкция. Чтобы избежать тавтологии ("деструктор вызывается для деструкции объекта"), я по возможности старался не применять этот термин. Можно также сказать "когда объект выходит из области видимости". Локальный объект выходит из области видимости, когда функция, создавшая его, доходит до команды return. Глобальный или статический объект выходит из области видимости, когда прекращается работа программы.

Если уничтожается более одного объекта, деструкторы вызываются в порядке, обратном вызову конструкторов. То же касается и случая, когда уничтожаемый объект содержит объекты-члены. Далее приведена программа TutorPair, использующая деструкторы.

```
#include <stdio.h>
#include <iostream.h>
class Student
{
public:
    Student()
    {
        cout << "Конструируем Student\n";
        semesterHours = 0;
        gpa = 0.0;
    }
    ~Student()
    {
        cout << "Уничтожаем Student\n";
    }
    // ...остальные открытые члены...
protected:
    int:  semesterHours;
};  float gpa;
```

```
class Teacher
{
public:
    {
    Teacher()
    }
        cout << "Конструируем Teacher\n";
    {
~Teacher()
    }
};      cout << "Уничтожаем Teacher\n";

class TutorPair
{
public:
    TutorPair()
    {
        cout << "Конструируем Tutor?air\n";

    }   noMeetings = 0;

    ~TutorPair()

    }   cout << "Уничтожаем TutorPairW;

protected:
    Student s;
    Teacher t;
    int noMeetings;
};
int main(int argcs, char* pArgs[])
{
    TutorPair tp;
    cout << "Возвращаемся в main()\n";
    return 0;
}
```

Если вы запустите эту профамму, на экран будуг выведены следующие сообщения:

```
Конструируем Student
Конструируем Teacher
Конструируем TutorPair
Возвращаемся в main()
Уничтожаем TutorPair
Уничтожаем Teacher
Уничтожаем Student
```

Как видите, конструктор класса TutorPair вызывается при объявлении tp, а деструктор — перед завершающей скобкой функции main(). Порядок вызова деструкторов обратен порядку вызова конструкторов.

Глава 18

Аргументация конструирования

В этой главе...

- Как снабдить конструктор аргументами
- Перегрузка конструктора
- Определение конструкторов по умолчанию
- Конструирование членов класса
- Управление последовательностью конструирования

Класс представляет тип объекта в реальном мире. Например, мы использовали класс Student для представления студента и его свойств.

Точно так же, как и студенты, классы считают себя абсолютно самостоятельными. Однако, в отличие от студентов, класс действительно сам "ухаживает" за собой — он должен все время поддерживать себя в приемлемом состоянии. Например, отрицательный идентификационный номер студента не приемлем для объекта класса student. И только сам класс отвечает за то, что на этапе создания объекта номер инициализирован подходящим значением.

С++ позволяет профаммисту определить специальную функцию-член, которая называется конструктором и вызывается автоматически на этапе создания объекта. Именно конструктор позволяет классу правильно инициализировать объект во время его создания.

Конструкторы, приведенные в главе 17, "Создание и удаление объектов: конструктор и деструктор", не имеют аргументов, т.е. не умеют ничего, кроме инициализации объекта одним и тем же состоянием. В данной главе рассмотрены конструкторы с аргументами.

fauc снабдить конструктор аргументами

С++ позволяет программисту определить конструктор с аргументами, например:

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student(char *pName)
    {
        cout << " Конструируем студента " << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) — 1] = '\0';
    }
    // ...остальные открытые члены...
protected:
    char  name[40];
    int   semesterHours;
    float gpa;
};
```

Зачем конструкторам нужны аргументы

Возможность добавления аргументов к конструктору не требует особой, простите за каламбур, аргументации, но я все же приведу несколько аргументов в пользу применения аргументов. Во-первых, их использование в конструкторе достаточно удобно. Было бы несколько непоследовательно требовать от программиста сначала конструировать объект, а затем вызывать инициализирующую функцию с тем, чтобы она проводила инициализацию, специфичную для данного объекта. Конструктор, поддерживающий аргументы, похож на магазин при автозаправке: он предоставляет полный сервис.

Другая, более важная причина использования аргументов в конструкторе состоит в том, что иногда это единственный способ создать объект с необходимыми начальными значениями. Вспомните, что работа конструктора заключается в создании корректного (в смысле требований данного класса) объекта. Если какой-то созданный по умолчанию объект не отвечает требованиям программы, значит, конструктор не выполняет свою работу.

Например, банковский счет без номера не является приемлемым (C++ все равно, каков номер счета, но это почему-то волнует банк). Можно создать объект BankAccount без номера, а затем потребовать от приложения вызвать некоторую функцию-член для инициализации номера счета перед использованием. Однако это нарушает наши правила, поскольку при таком подходе класс вынужден полагаться на то, что эти действия будут выполнены внешним приложением.

Как использовать конструктор с аргументами

Идея использования аргументов проста. Как известно, функции-члены могут иметь аргументы, поэтому конструктор, будучи функцией-членом, тоже может иметь аргументы.

При этом нельзя забывать, что вы вызываете конструктор не как нормальную функцию и передать ему аргумент можно только в момент создания объекта. Так, приведенная ниже программа создает объект s класса student, вызывая конструктор Student(char*). Объект s уничтожается в момент возврата из функции main().

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student(char *pName)
    {
        cout << "Конструируем студента " << pName << "\n";
        strncpy(name, pNarrLe, sizeof(name) ) ;
        name[sizeof(name) - 1]  = '\0';
        semesterHours = 0;
        gpa = 0.0;
    }
    ~Student()
    {
        cout << "Ликвидируем " << name << "\n";

        // неплохо бы стереть имя уничтожаемого студента,
        name[0] = '\0';
    }

    // ...остальные открытые члены...
protected:
    char  name[40];
    int   semesterHours;
    float gpa;
```

```
};

int main(int argcs, char* pArgs[])
{
    Student s("Danny"); // Создаем Дэкни
    return 0;
)                          //а теперь избавимся от него
```

В этом примере конструктор выглядит почти так же, как и конструктор из главы 17, "Создание и удаление объектов: конструктор и деструктор", с тем лишь отличием, что он принимает аргумент pName, имеющий тип char*. Этот конструктор инициализирует все данные-члены нулевыми значениями, за исключением члена name, который инициализируется строкой pName.

Объект s создается в функции main (). Аргумент, передаваемый конструктору, находится в строке объявления s сразу же за именем объекта. Благодаря такому объявлению студент s получил имя Danny. Закрывающая фигурная скобка функции main() вызывает гром и молнию деструктора на голову несчастного Danny.

Запуск этой программы выведет на экран следующее:

```
Конструируем студента Danny
Ликвидируем Danny
```



Многие конструкторы в этой главе нарушают правило "функции размером более трех строк не должны быть inline-функциями". Я просто решил облегчить вам чтение (а теперь — ваши аплодисменты!).

Конструкторы и деструкторы могут быть реализованы также следующим образом:

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student(char *pName)
    ~Student()
    //...остальные открытые члены...
protected:
    char  name[40];
    int   semesterHours;
    float gpa;
};

// определим функции (обратите внимание на
// отсутствие типа возвращаемого значения)
Student::Student(char *pName)
{
    cout << "Конструируем студента " « pName << "\n";
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
    semesterHours = 0;
    gpa =0.0;
}
//обратите внимание, как выглядит деструктор
Student::~Student()

    cout << "Ликвидируем " << name << "\n";
}
```

Когда вы поднакопите побольше опыта в С++, то будете легко переводить функции из одной формы записи в другую.

Перегрузка конструктора

Поскольку в этой главе проводятся параллели между конструктором и обычными функциями-членами, я позволю себе еше одну параллель: конструкторы можно перегружать.



Словосочетание "перегруженная функция" означает, что определено несколько функций с одинаковым именем, но разными типами аргументов. Если вы немного подзабыли этот термин, освежите память. обратившись к главе 6, "Создание функций".

С++ выбирает вызываемый конструктор, исходя из аргументов, передаваемых при объявлении объекта. Например, класс Student может одновременно иметь три конструктора, что продемонстрировано в следующем примере:

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student()
    i
        cout << "Создаем студента без имени\n";
        semesterHours = 0;
        gpa = C.O;
        name[0] = '\0';
    }
    Student(char *pName)
    {
        cout << "Создаем студента " << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\0';
        semesterHours = 0;
        gpa = 0;
    }
    Student(char *pName, int xfrHours, float xfrGPA)
    {
        cout << "Создаем студента " << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\Q';
        semesterHours = xfrHours;
        gpa = xfrGPA;
    }
    ~Student()
    {
        cout << "Ликвидируем студента\n";
    }
    // ...остальные открытые члены...
protected:
    char  name[40];
    int   semesterHours;
    float gpa;
};

// приведенный ниже фрагмент по очереди
// вызывает каждый из конструкторов
int main(int argcs, char* pArgs[])
{
    Student noName;
```

```
    Student freshMan("Smell E. Fish");
    Student xfer("Upp R. Classman", 80, 2.5);
    return 0;
}
```

Поскольку объект noName реализован без аргументов, он конструируется с использованием Student::Student(), который называется конструктором по умолчанию или пустым конструктором. (Я предпочитаю последнее название, но, поскольку первое более распространенное, в этой книге будет использоваться именно оно.) Объект freshMan создается с использованием конструктора, которому нужен только один аргумент типа char*; объекту xfer требуется конструктор с тремя аргументами.

Заметьте, что все три конструктора (и особенно два последних) очень похожи. Добавив значения по умолчанию в последний конструктор, все три можно объединить в один, что и сделано в приведенном ниже коде.

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    Student(char *pName ="no name",
            int xfrHours = 0,
            float xfrGPA = 0.0)
    {
        cout << "Создаем студента " << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) — 1] = '\0';
        semesterHours = xfrHours;
        qpa = xfrGPA;
    }
    ~Student()
    {
        cout << "Ликвидируем студента\п";
    }
    // ...остальные открытые члены...
protected:
    char  name[40];
    int   semesterHours;
    float gpa;
};

int main(int argcs, char* pArgs[])
f
    Student noName;
    Student freshMan("Smell E. Fish");
    Student xfer("Upp R. Classman", 80, 2.5);
    return 0;
}
```

Теперь все три объекта строятся с помощью одного и того же конструктора, а значения по умолчанию используются для аргументов, отсутствующих в объектах freshMan и noName.



В ранних версиях C++ вы не смогли бы создать конструктор по умолчанию, предусмотрев значения по умолчанию для всех аргументов. Конструктор по умолчанию должен был быть определен явно. Так что будьте готовы к тому, что некоторые старые версии компиляторов могут потребовать явного определения конструктора по умолчанию.

Определение конструкторов по умолчанию

Стоит отметить, что в C++ каждый класс должен иметь свой конструктор. Казалось бы, C++ должен генерировать сообщение об ошибке в случае, когда класс не оснащен конструктором, однако этого не происходит. Дело в том, что для обеспечения совместимости с существующим кодом С, который ничего не знает о конструкторах, C++ автоматически создает конструктор по умолчанию (так сказать, умалчиваемый конструктор по умолчанию), который инициализирует все данные-члены объекта нулями.

Если ваш класс имеет конструктор, C++ не будет автоматически его создавать (как только C++ убеждается в том, что это не программа на С, он снимает с себя всю ответственность по обеспечению совместимости).



Вывод: если вы определили конструктор для вашего класса и при этом хотите, чтобы класс имел конструктор по умолчанию, то должны явно определить такой конструктор сами.

Привсденный ниже фрагмент демонстрирует сказанное. Этот пример вполне корректен.

```
class Student
{
  //...то же, что и раньше, только без конструкторов
};
int main(int argcs, char* pArgs[])
{
    Student noName;
    return 0;
}
```

Поскольку noName объявлен без аргументов, для создания этого объекта C++ вызывает конструктор по умолчанию. Однако в этом примере программист не определил никаких конструкторов для класса student и C++ сам создает конструктор по умолчанию, который просто инициализирует все данные-члены объекта нулевыми значениями.

Приведенный далее пример компилятор с негодованием отвергнет.

```
class Student
{
public:
    Student(char *pName);
};
int main(int argcs, char* pArgs[])
{
    Student noName;
    return 0;
}
```

То, что здесь добавлен конструктор student (char*), выглядит безобидно, но при этом заставляет C++ отказаться от автоматической генерации конструктора по умолчанию. Поэтому компиляция данного кода в GNU C++ вызовет сообщение об ошибке (оно может быть различным у разных компиляторов, но смысл его один: у класса не определен конструктор по умолчанию):

```
Error: no matching function for call Student::Student()
 (Ошибка: не найдена функция, соответствующая
         вызову Student::Student())
```

Компилятор сообщает, что он не может найти конструктор student: : Student (). Добавив конструктор по умолчанию, можно решить эту проблему:

```
class Student
{
public:
    Student(char *pName);
    Student();
};
int main(int argcs, char* pArgs[])
{
    Student noName;
    return 0;
}
```

Это пример нелогичности, поясняющий, за что программистам на C++ платят большие деньги.

Конструирование членов класса

В предыдущих примерах использовались данные-члены простых типов, такие, как float или int. Переменные таких простых типов легко инициализировать, передав необходимое значение конструктору. Но что, если класс содержит данные-члены, которые являются объектами других классов? Рассмотрим приведенный ниже пример.

```
#include <iostream.h>
#include <string.h>

int nextStudentId = 0;
class StudentId
(
public:
    StudentId()
    {
        value = ++nextStudentId;
        coat << "Присваиваем студенту идентификатор "
             << value << "\n";
    }
protected:
    int value;
};

class Student
{
public:
    Student(char *pName = "no name")
    {
        cout << "Создаем студента " << pName << "\n";
        :;trncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\0';
    }
protected:
    char name[40];
    Studentld id;
};

int main(int argcs, char* pArgs[])
```

```
{
    Student s("Randy");
    return 0;
}
```

В момент создания объекту типа Student присваивается собственный идентификатор. В данном примере идентификаторы "раздаются" последовательно. с помощью глобальной переменной nextStudentID.

Наш класс Student содержит член id. который является экземпляром класса StudentID. Конструктор класса Student не может присвоить значение члену id, поскольку Student не имеет доступа к защищенным членам класса St.uder.tlD. Можно было бы сделать student другом класса StudentID, но такой подход нарушил бы положение объектно-ориентированного программирования, утверждающее, что каждый класс должен заниматься своим делом. Нам нужна возможность вызывать конструктор класса StudentID в процессе создания класса Student.

С++ делает это автоматически, инициализируя член id с помощью конструктора по умолчанию StudentID:: StudentiD (). Это происходит после вызова конструктора класса Student, но до того, как управление передается первой строке этого конструктора. (Выполните в пошаговом режиме приведенную выше программу, и вы поймете, о чем я говорю. Только не забудьте при компиляции включить опцию Outline inline functions.) Выполнение приведенной выше программы выведет на экран следующие строки:

```
Присваиваем студенту идентификатор 1
Создаем студента Randy
```

Обратите внимание: сообщение от конструктора StudentiD появилось раньше, чем сообщение от конструктора Student.

(Поскольку у нас все конструкторы выполняют вывод на экран, вы можете решить, что они всегда должны поступать подобным образом. На самом деле подавляющее большинство конструкторов работают "молча".)

Если программист не обеспечит свой класс конструктором, то конструктор по умолчанию, созданный С++, вызовет конструкторы всех данных-членов для их инициализации. То же касается и уничтожения объекта. Деструктор класса автоматически вызывает деструкторы всех данных-членов (которые, само собой, его имеют).

Теперь мы знаем, что будет с конструктором по умолчанию. Но что, если мы захотим вызвать другой конструктор? Куда в этом случае нужно поместить объект? Вот что я имею в виду: представьте себе, что вместо автоматической генерации идентификатора студента, необходимо передать его конструктору student с тем, чтобы он, в свою очередь, передал его конструктору StudentID.

Для начала я покажу вам способ, который работать не будет.

```
#include <iostream.h>
#include <string.h>

class StudentId
{
public:
    StudentId(int id = 0)
    {
        value = id;
        cout << "Присваиваем студенту id, равный "
             << value << "\n";
    }
    ~StudentId()
    {
        cout << "Удаляем студента " << value << "\n";
    }
```

```
protected:
    int value;
};

class Student
{
public:
    Student(char *pName = "no name", int ssId = 0)
    {
        cout « "Создаем студента " << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\0';
        // Вот это можно и не пытаться делать
        // - толку не будет
        StudentId id(ssId);
    }
protected:
    char name[40];
    StudentId id;
};

int main(int argcs, char* pArgs[])
{
    Student s("Randy", 1234);
    cout « "Сообщение из main\n";
    return 0;
}
```

Конструктор класса `StudentID` был переписан так, чтобы он мог принимать внешнее значение (значение по умолчанию необходимо для того, чтобы приведенный фрагмент откомпилировался без ошибок, которые в противном случае появятся; почему — станет понятно чуть позже). Внутри конструктора `Student` программист (т.е. я) попытался невиданным доселе способом сконструировать объект id класса `StudentID`.

Если вы внимательно посмотрите на сообщения, которые выдаются в результате работы этой программы, то поймете, в чем проблема,

```
Присваиваем студенту id, равный 0
Создаем студента Randy
Присваиваем студенту id, равный 1234
Удаляем студента 1234
Сообщение из main
Удаляем студента 0
```

Первая проблема заключается в том, что конструктор класса `StudentID` вызывается дважды: в первый раз с нулем и только затем с ожидаемым числом 1234. Кроме того, объект с идентификатором 1234 ликвидируется перед выводом сообщения от main{). Очевидно, объект класса studentlD ликвидируется внутри конструктора класса Student.

Объяснить такое странное поведение программы довольно просто. Член id уже существует к моменту перехода управления к телу конструктора `Student`. Поэтому вместо инициализации уже существующего члена id объявление в последней строке конструктора `Student` вызывает создание локального объекта с таким же именем. Этот локальный объект и уничтожается при выходе из конструктора.

Очевидно, нужен некий механизм конструирования не нового объекта, а уже существующего. Этот механизм должен работать перед открытием фигурной скобки конструктора. Для этого в C++ определена следующая конструкция:

```
    Student(char *pName = "no name", int ssId = 0):id(ssId)
    {
        cout « "Создаем студента " << pName « "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\0';
    }
protected:
    char name[40];
    StudentId id;
);
```

Обратите особое внимание на первую строку конструктора. В этой строке есть кое-что, с чем вы до этого не встречались. Следующая за двоеточием команда вызывает конструкторы членов данного класса. Компилятор C++ прочтет эту строку так: "Вызвать конструктор для члена id с аргументом ssId. Все остальные данные-члены, не вызванные явно, строить с использованием конструктора по умолчанию".

Результат работы этой программы свидетельствует, что все получилось так, как мы и хотели.

```
Присваиваем студенту id, равный 1234
Создаем студента Randy
Сообщение из main
Удаляем студента 1234
```

Символ ":" можно использовать и для присвоения значений константам или членам ссылочного типа. Это можно сделать так, как показано в приведенном ниже примере.

```
class SillyClass
{
public:
    SillyClass(int& i) : ten(10), refI(i)
    {
    }
protected:
    const int ten;
    int& refI;
};
int main(int argcs, char* pArgs[])
{
    int i;
    SillyClass sc(i);
    return 0;
}
```

В момент входа в тело конструктора SillyClass данные-члены ten и refI уже созданы. Это аналогично объявлению константы и ссылочной переменной в теле функции. Таким переменным значение должно быть присвоено в момент объявления, и другим способом их инициализации в классе вам добиться не удастся.

Не попадитесь в ловушку

Еще раз взгляните на объявление объектов класса Student из приведенного выше примера:

```
Student noName;
Student freshMan("Smell E. Fish");
Student xfer("Upp R. Classman", 80, 2.5);
```

Все объекты типа Student, за исключением noName, объявлены со скобками, в которых находятся передаваемые классу аргументы. Почему же объект noName объявлен без скобок?

С точки зрения приверженцев последовательности и аккуратности, лучше было бы объявлять этот объект так:

```
Student noName();
```

Конечно, можно сделать и так, но это не приведет к ожидаемому результату. Вместо объявления объекта noName, создаваемого с помощью конструктора по умолчанию для класса Student, будет объявлена функция, возвращающая по значению объект класса student. Мистика!

Приведенные ниже два объявления демонстрируют, как похожи объявления объекта и функции в формате С++. (Я-то считаю, что это можно было сделать и по-другому, но кто будет со мной считаться?..) Единственное отличие заключается в том, что при объявлении функции в скобках стоят названия типов, а при объявлении объекта в скобках содержатся объекты.

```
Student thisIsAFunc(int);
Student thisIsAnObject(10);
```

Если скобки пусты, невозможно однозначно сказать, что объявляется — функция или объект. Для обеспечения совместимости с С в С++ считается, что объявление с пустыми скобками является объявлением функции (более надежной альтернативой было бы требование наличия ключевого слова void при объявлении функции, но тогда нарушалось бы условие совместимости с существующими программами на С...).

Управление последовательностью конструирования

При наличии нескольких объектов, в которых определены конструкторы, программист обычно не заботится о том, в каком порядке будут конструироваться эти объекты. Однако, если один или несколько конструкторов имеют побочное действие, различная последовательность конструирования может привести к разным результатам.

Порядок создания объектов подчиняется перечисленным ниже правилам.

- ✓ Локальные и статические объекты создаются в том порядке, в котором они объявлены в программе.
- ✓ Статические объекты создаются только один раз.
- ✓ Все глобальные объекты создаются до вызова функции main().
- ✓ Нет какого-либо определенного порядка создания глобальных объектов.
- ✓ Члены создаются в том порядке, в котором они объявлены внутри класса.
- ✓ Деструкторы вызываются в порядке, обратном порядку вызова конструкторов.



Статическая переменная — это переменная, которая является локальной по отношению к функции, но при этом сохраняет свое значение между вызовами функции. Глобальная переменная — это переменная, объявленная вне какой-либо функции.

Рассмотрим каждое из приведенных выше правил.

Локальные объекты создаются последовательно

Локальные объекты создаются в том порядке, Б котором в программе встречаются их объявления- Обычно это порядок появления кода объявлений в функции. (Если, конечно, в функции нет безусловных переходов, "перепрыгивающих" через объявления. Кстати говоря, безусловные переходы между объявлениями лучше не использовать — это затрудняет чтение и компиляцию программы.)

Статические объекты создаются один раз

Статические переменные подобны обычным локальным переменным с тем отличием, что они создаются только один раз. Это очевидно, поскольку статические переменные сохраняют свое значение от вызова к вызову функции. В отличие от С, который может инициализировать статическую переменную в начале программы, C++ дождется, когда управление перейдет строке с объявлением статической переменной, и только тогда начнет ее создание. Разберемся в приведенной ниже простой программе.

```
#include <iostream.h>
#include <string.h>
class DoNothing
{
public:
    DoNothing(int initial)
    {
        cout << "Переменная DoNothing создана со значением "
             << initial
             << "\n";
    }
};
void fn(int i)
{
    static DoNothing dn(i);
    cout << "Мы - в функции fn с i = " << i << "\n";
}

int main(int argcs, char* pArgs[])
{
    fn(10);
    fn(20);
    return 0;
}
```

После запуска этой программы на экране появится следующее:

```
Переменная DoNothing создана со значением 10
Мы - в функции fn с i = 10
Мы - в функции fn с i = 20
```

Обратите внимание, что сообщение от функции fn() появилось дважды, а сообщение от конструктора DoNothing — только при первом вызове fn().

Все глобальные объекты создаются до вызова main()

Все глобальные объекты входят в область видимости программы. Таким образом, все они конструируются до того, как управление передается функции main().



При отладке такой порядок может привести к неприятностям. Некоторые отладчики пытаются выполнить весь код, который находится до main O, и только потом передать управление пользователю. Это прекрасно подходит для C, поскольку до входа в функцию main() там не может быть никакого кода, написанного пользователем. Однако в C++ это может стать причиной большой головной боли, поскольку тела конструкторов для всех глобальных объектов к моменту передачи управления main() уже выполнены. Если хоть один из этих конструкторов содержит серьезный "жучок", программа погибнет до того, как начнет выполняться!

Существует несколько подходов к решению этой проблемы. Первый заключается в том, чтобы проверять каждый конструктор на локальных объектах перед тем, как использовать его для глобальных. Если это не поможет решить проблему, можно попытаться добавить команды вывода сообщений в начало всех конструкторов, которые, по вашему предположению, могут иметь ошибки. Последнее сообщение, которое вы увидите, вероятно, будет сообщением конструктора с ошибкой.

Порядок создания глобальных объектов не определен

Локальные объекты создаются в порядке выполнения программы. Для глобальных же объектов порядок создания не определен. Как вы помните, глобальные объекты входят в область видимости программы одновременно. Возникает вопрос: почему бы тогда компилятору не начать с начала файла с исходной программой и не создавать глобальные объекты в порядке их объявления? (Честно говоря, я подозреваю, что на самом деле большинство компиляторов так и поступают.) Увы, такой подход отлично работал бы, но только в том случае, если бы программа всегда состояла из одного файла.

Однако большинство программ в реальном мире состоят из нескольких файлов, которые компилируются каждый в отдельности, а уже затем связываются в единое целое. Поскольку компилятор не управляет порядком связывания, он не может влиять на порядок вызова конструкторов глобальных объектов в разных файлах.

В принципе в большинстве случаев порядок создания глобальных объектов не так уж и важен. Тем не менее иногда это может привести к ошибкам, которые потом очень сложно отследить (такое случается довольно часто, чтобы обратить на это внимание в книге).

Разберем приведенный ниже пример.

```
//в файле Student.H:
class Student
{
public:
    Student (unsigned id) : studentId(id)
    {
    }
    const unsigned Studentld;
};
class Tutor
{
public:
    Tutor (Students s)
    {
        tutoredId = s.studentId;
    i
protected:
    unsigned tutoredld;
};
//в файле FILE1.CPP
//Создаем студента
```

```
Student randy(1234);

//в файле FILE2.CPP
//Назначаем студенту учителя
Tutor jenny(randy);
```

В этом примере конструктор student присваивает студенту идентификатор, а конструктор класса Tutor записывает этот идентификатор студента, которому нужен учитель. Программа объявляет студента randy, а затем назначает ему учителя jenny.

При этом подразумевается, что randy создается раньше, чем jenny; в этом-то и состоит проблема. Представьте себе, что порядок создания этих объектов будет другим. Тогда объект jenny будет построен с использованием блока памяти, который пока что не является объектом типа Student, а значит, вместо идентификатора студента в randy будет находиться непредсказуемое значение.



Приведенный выше пример несложен и несколько надуман. Однако проблемы, создаваемые глобальными объектами, могут оказаться гораздо коварнее. Во избежание этого не допускайте, чтобы конструктор глобального объекта обращался к другому глобальному объекту.

Члены создаются в порядке их объявления

Члены класса создаются в соответствии с порядком, в котором они объявлены внутри класса. Это не так просто и очевидно, как может показаться на первый взгляд. Рассмотрим пример.

```
class Student
{
public:
    Student (unsigned id, unsigned age)  : sAge(age), sId(id)
    {
    }
    const unsigned sId;
    const unsigned sAge;
};
```

В этом примере sId создается до sAge, несмотря на то что он стоит вторым в инициализирующем списке конструктора. Впрочем, единственный случай, когда можно заметить какую-то разницу в порядке конструирования, — это когда оба члена класса имеют конструкторы, которым присуще какое-либо общее побочное действие.

Деструкторы удаляют объекты в порядке, обратном порядку их создания

В каком бы порядке не вызывались конструкторы объектов, вы можете быть уверены, что их деструкторы будут вызваны в обратном порядке. (Приятно сознавать, что хоть одно правило в C++ не имеет никаких "или", "и" либо "но".)

Глава 19

Копирующий конструктор

В этой главе...

- ✓ Копирование объекта
- ✓ Автоматический конструктор копирования
- ✓ "Мелкие" и "глубокие" копии
- ✓ Временные объекты

Конструктор — это специальная функция, которая автоматически вызывается C++ при создании объекта с тем, чтобы предоставить ему возможность проинициализировать самого себя. В главе 17, "Создание и удаление объектов: конструктор и деструктор", описаны основные концепции применения конструкторов, в главе 18, "Аргументация конструирования", вы познакомились с разными типами конструкторов. А в настоящей главе рассматривается частный случай, известный под названием копирующего конструктора (или конструктора копирования).

Копирование объекта

Конструктор, который используется C++ для создания копий объекта, называется копирующим конструктором. Он имеет вид `X::X(X&)` (или `x::X(const X&)`), где X — имя класса. Да, это не ошибка — это действительно конструктор класса x, который требует в качестве аргумента ссылку на объект класса X. Я понимаю, что это звучит несколько бессмысленно, но не торопитесь с выводами и позвольте объяснить, зачем такое "чудо" в C++.

Зачем это нужно

Подумайте о том, что будет происходить в программе, если вы вызовете следующую функцию:

```
void fn(Student fs)
{
    //тот же сценарий с другими аргументами
}
int main(int argcs, char* pArgs[])
{
    Student ms;
    fn(ms);
    return 0;
}
```

При вызове описанной функции fn () ей будет передан в качестве аргумента не сам объект, а его копия.

Теперь попробуем понять, что же значит — создать копию объекта. Для этого требуется конструктор, который будет создавать объект (даже если копируется уже существующий объект). C++ мог бы побайтово скопировать существующий объект в но-

вый, но как быть, если побайтовая копия не совсем то, что нам нужно? Что, если мы хотим нечто иное? (Не спрашивайте у меня пока, что такое это "иное" и зачем оно нужно. Немного терпения!) У нас должна быть возможность самим определять, как будет создаваться копия объекта.



В C++ аргументы функции передаются по значению.

Таким образом, в приведенном выше примере необходим копирующий конструктор, который будет выполнять копирование объекта ms при вызове функции fn(). Этот частный копирующий конструктор и есть Student::Student(Student&) (попробуйте-ка произнести эту скороговорку...).

Использование конструктора копирования

Лучший путь понять, как работает конструктор копирования, — это увидеть его в действии. Рассмотрим приведенный ниже пример с классом Student.

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    //обычный конструктор
    Student(char *pName = "no name", int ssId = 0)
    {
        cout << "Создаем нового студента "
             << pName
             << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\C';
        id = ssId;
    }

    //Копирующий конструктор
    Student(Student& s)
    {
        cout << "Конструируем копию "
             << s.name
             << "\n";
        strcpy(name, "Копия ");
        strcat(name, s.name);
        id = s.id;
    }

    ~Student()
    {
        cout << "Ликвидируем: " << name << "\n";
    }

protected:
    char name[40];
    int id;
};

// fn принимает аргумент по значению
void fn(Student s)
```

```
{
    cout << "Сообщение из функции fn()\n";
}

int main(int argcs, char* pArgs[])
{
    Student randy("Randy", 1234);
    cout << "Вызываем функцию fn()\n";
    fn(randy);
    cout << "Вернулись из fn()\n";
    return 0;
}
```

После запуска этой программы на экран будут выведены следующие строки:

```
Создаем нового студента Randy
Вызываем функцию fn()
Конструируем копию Randy
Сообщение из функции fn()
Ликвидируем: Копия Randy
Вернулись из fn()
Ликвидируем: Randy
```

Давайте внимательно рассмотрим, как же работает эта программа. Обычный конструктор выводит первую строку; затем main() выводит строку "Вызываем. . .". После этого C++ вызывает копирующий конструктор для создания копии объекта randy (которая и передается функции fn () в качестве аргумента). Эта копия будет ликвидирована при возврате из функции fn (); исходный же объект randy ликвидируется при выходе ИЗ main () .

Копирующий конструктор выглядит как обычный, но обладает особенностью получать в качестве аргумента ссылку на другой объект того же класса. (Обратите внимание, что использованный в примере копирующий конструктор, помимо простого копирования объекта, делает кое-что еще, например выводит строку "Конструируем копию. . .". Эту возможность выполнять кроме собственно копирования и другие действия можно будет с успехом применить для решения разных задач. Конечно, копирующие конструкторы обычно ограничиваются созданием копий уже существующих объектов, но на самом деле они могут делать все, что угодно программисту.)

Автоматический конструктор копирования

Копирующий конструктор важен ничуть не менее конструктора по умолчанию. Важен настолько, что C++ считает невозможным существование класса без копирующего конструктора. Если вы не создадите свою версию такого конструктора, C++ создаст ее за вас. (Это несколько отличается от конструктора по умолчанию, который создается только в том случае, если в вашем классе не определено вообще никаких конструкторов.)

Копирующий конструктор, создаваемый C++, выполняет поэлементное копирование всех членов-данных. Ранее копирующий конструктор, создаваемый C++, выполнял побитовое копирование. Отличие между этими методами заключается в том, что при поэлементном копировании для каждого члена класса вызываются соответствующие копирующие конструкторы (если они существуют), тогда как при побитовом копировании конструкторы не вызывались. Разницу в результатах можно увидеть, выполнив приведенный пример.

```
#include <iostream.h>
#include <string.h>

class Student
{
public:
    Student(char *pName = "no name")
    {
        cout << "Создаем нового студента "
             << pName << "\n";
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\0';
    }

    Student(Student& s)
    {
        cout << "Создаем копию от " << s.name << "\n";
        strcpy(name, "Копия ");
        strcat(name, s.name);
    }

    ~Student()
    {
        cout << "Ликвидируем " << name << "\n";
    }

protected:
    char name[40];
};

class Tutor
{
public:
    Tutor(Student& s) : student(s)
        // вызываем копирующий конструктор
        // для члена student
    {
        cout << "Создаем учителя\n";
    }
protected:
    Student student;
};

void fn(Tutor tutor)
{
    cout << "Это сообщение из функции fn()\n";
}

int main(int argcs, char* pArgs[])
{
    Student randy("Randy");
    Tutor tutor(randy);
    cout << "Вызываем функцию fn()\n";
    fn(tutor);
    cout << "Вернулись из функции fn() \n";
    return 0;
}
```

Запуск этой программы приведет к выводу таких сообщений:

```
Создаем нового студента Randy
Создаем копию от Randy
Создаем учителя
Вызываем функцию fn()
Создаем копию от Копия Randy
Это сообщение из функции fn()
Ликвидируем Копия Копия Randy
Вернулись из функции fn()
Ликвидируем Копия Randy
Ликвидируем Randy
```

Конструирование объекта randy приводит к вызову конструктора Student, который выводит первое сообщение.

Объект tutor создается с использованием конструктора Tutor(Students). Этот конструктор инициализирует член Tutor::student, вызывая копирующий конструктор для Student. Он и выводит вторую строку с сообщением.

Вызов функции fn() требует создания копии tutor. Поскольку я не обеспечил Tutor копирующим конструктором, каждый член объекта tutor копируется посредством конструктора, созданного С++ по умолчанию. При этом, как отмечалось ранее, для копирования члена tutor.student вызывается конструктор копирования класса Student.

"Мелкие" и "глубокие" копии

Выполнение поэлементного копирования — естественная задача конструктора копирования. Но что еще можно сделать с помощью такого конструктора? Когда наконец можно попытаться сделать что-то поинтереснее, чем программирование поэлементного копирования и объединения каких-то строк с именем несуществующего студента?

Представим ситуацию, когда конструктор распределяет объекту некоторые системные ресурсы, например память из кучи. Если копирующий конструктор будет выполнять простое копирование без выделения памяти из кучи для копируемого объекта, может возникнуть ситуация, когда два объекта будут считать, что именно они являются владельцами одного блока памяти. Ситуация еще более усугубится при вызове деструкторов обоих объектов, которые попытаются освободить одну и ту же память. Взгляните на приведенный ниже пример.

```
#include <iostream.h>
#include <string.h>

class Person
{
public:
    Person(char *pN)
    {
        cout << "Создаем " << pN << "\n";
        pName = new char[strlen(pN) + 1];
        if (pName != 0)
        {
            strcpy(pName, pN);
        }
    }
    ~Person()
    {
        cout << "Ликвидируем " << pName << "\n";
        // в качестве действия деструктора сотрем имя
```

```
        pName[0] = '\0';
        delete pName;
    }
protected:
    char *pName;
};

int main(int argcs, char* pArgs[])
{
    Person p1 ("Randy");
    Person p2 = p1;    // Вызов копирующего конструктора
    return 0;
}
```

В этом примере конструктор для Person выделит память из кучи для хранения в ней имени произвольной длины, что невозможно при использовании массивов. Деструктор возвращает эту память в кучу. Основная программа создает объект p1, описывающий человека, после чего создается копия этого объекта — p2.

После запуска этой программы вы получите сообщение только от одного конструктора. Это неудивительно, поскольку копия p2 создается с помощью предоставляемого C++ конструктора копирования по умолчанию, а он не выводит никаких сообщений. Однако, после того как p1 и p2 выходят из области видимости, вы не получите двух сообщений о ликвидации объектов, как можно было ожидать.

Более того, если вы выполните эту программу в пошаговом режиме в каком-либо отладчике, то получите сообщение об ошибке. Что же происходит?

Конструктор вызывается один раз и выделяет блок памяти из кучи для хранения в нем имени человека. Копирующий конструктор, создаваемый C++, просто копирует этот адрес в новый объект, без выделения нового блока памяти.

Когда объекты ликвидируются, деструктор для p2 первым получает доступ к этому блоку памяти. Этот деструктор стирает имя и освобождает блок памяти. К тому времени как деструктор p1 получает доступ к этому блоку, память уже очищена, а имя стерто. Теперь понятно, откуда взялось сообщение об ошибке. Суть проблемы проиллюстрирована на рис. 19.1. Объект p1 копируется в новый объект p2, но не копируются используемые им ресурсы. Таким образом, p1 и p2 указывают на один и тот же ресурс (в данном случае это блок памяти). Такое явление называется "мелким" (shallow) копированием, поскольку при этом копируются только члены класса как таковые.

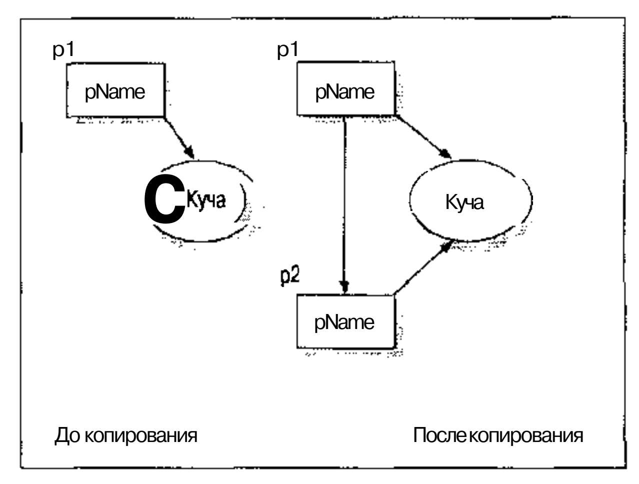


Рис. 19.1. Мелкое копирование объекта p1 в p2

Для решения указанной проблемы нужен такой копирующий конструктор, который будет выделять ресурсы для нового объекта. Давайте добавим такой конструктор к классу и посмотрим, как он работает.

```
class Person
 (
public:
    //копирующий конструктор выделяет новый
    //блок памяти из кучи
    Person(Person& p)
    {
        cout << "Копируем " <<p.pName
             << " в отдельный блок\n";
        pName = new char[strlen(p.pName) + 1];
        if (pName != 0)
        i
            strcpy(pName, p.pName);
        }
    }
    //...всеостальное оставляем без изменений...
```

Здесь копирующий конструктор выделяет новый блок памяти для имени, а затем копирует содержимое блока памяти исходного объекта в этот новый блок (рис. 19,2). Такое копирование называется "глубоким" (deep), поскольку копирует не только элементы, но и занятые ими ресурсы (конечно, аналогия, как говорится, притянута за уши, но ничего не поделаешь — не я придумал эти термины).

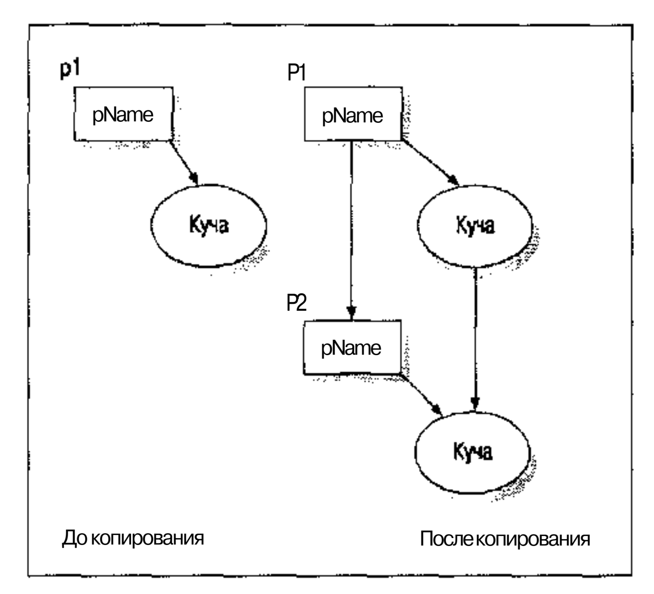


Рис. 19.2. Глубокое копирование p1 в p2

Запуск программы с новым копирующим конструктором приведет к выводу на экран следующих строк:

```
Создаем Randy
Копируем Randy в отдельный блок
Ликвидируем Randy
Ликвидируем Randy
```



Ресурсом, требующим глубокого копирования, является не только память в куче, но и открытые файлы, порты, выделенные аппаратные средства (например, принтеры) и т.п. Кроме того, все эти типы ресурсов должны освобождаться деструктором. Таким образом, можно вывести правило: глубокое копирование необходимо всегда, когда деструктор класса освобождает ресурсы.

Временные объекты

Копии создаются не только тогда, когда объекты передаются в функции по значению. Копии объектов могут создаваться и по другим причинам, например при возврате объекта по значению. Рассмотрим пример.

```
Student fn();    //возвращает объект по значению
int main (int argcs, char* pArgs[])
{
    Student s;
    s = fn();     //в результате вызова fn() будет
                  // создан временный объект
   return 0;
)
```

Функция fn() возвращает объект по значению. В конечном счете этот объект будет скопирован в s, но где он находится до этого?

Для хранения таких объектов C++ создает временные объекты (такие объекты создаются и в некоторых других случаях). "Хорошо, — скажете вы, — C++ создает временные объекты, но откуда он знает, когда их надо уничтожать?" (Спасибо за хороший вопрос!) В нашем примере это не имеет особого значения, поскольку временный объект выйдет из области видимости, как только копирующий конструктор скопирует его в s. Но что, если s будет определено как ссылка?

```
int main (int argcs, char* pArgs[])
 (
    Student& refS = fn();
    //...что теперь?...
    return 0;
}
```

Теперь период жизни временного объекта имеет большое значение, поскольку ссылка refs продолжает свое существование независимо от существования объекта! В приведенном ниже примере я отметил место, начиная с которого временный объект становится недоступен.

```
Student fn1();
int fn2(Student&);
int main (int argcs, char* pArgs[])
{
    int x;
    // Создаем объект Student, вызывая fn1(),
    // а затем передаем этот объект функции fn2().
    // fn2() возвращает целочисленное значение,
    // которое используется для выполнения
    // некоторых вычислений.
    // Весь этот период временный объект, возвращенный
    // функцией fnl () , доступен
    x = 3*fn2 (fnl () ) + Ю;
```

```
    // временный объект, который вернула функция fn1(),
    // становится недоступен
    //...остальной код...
    return 0;
}
```

Таким образом, пример с использованием ссылки неверен, поскольку объект выйдет из области видимости, а refS будет продолжать существовать и ссылка будет указывать на несуществующий объект.

Вы можете подумать, что изучение всего этого копирования объектов туда и обратно — пустая трата времени. Что, если вы не хотите делать все эти копии? Самое простое решение заключается в передаче и приеме объектов функции по ссылке. Это исключает все описанные неприятности.

Но как убедиться, что C++ не создает временных объектов незаметно для вас? Допустим, ваш класс использует ресурсы, которые вы не хотите копировать. Что же вам делать?

Можно просто использовать вывод сообщения в копирующем конструкторе, которое предупредит вас о том, что была сделана копия. А можно объявить копирующий конструктор защищенной функцией, как показано в приведенном ниже примере.

```
class Student
{
protected:
    Student(Student& s){}
public:
    //...все остальное как обычно...
};
```

Такой подход исключит использование копирующего конструктора любыми внешними функциями, включая сам C++, а значит, запретит создание копий ваших объектов student (позволяя при этом создавать копии функциям-членам).

Использование копирующего конструктора для создания временных объектов и копий объектов вызывает один интересный вопрос. Рассмотрим очередной пример.

```
class Student
{
public:
    Student()
    {
        //...все, что угодно...
    }
    Student(Student s)
    {
    }   //...все, что угодно...
};
void fn(Student fs)
{
}
int main (int argcs, char* pArgs[])
{
    Student ms;
    fn(ms);
    return 0;
}
```

И в самом деле, почему бы не объявить копирующий конструктор класса Student как Student::Student(Student)? Однако такое объявление попросту невозможно!

При попытке скомпилировать программу с таким объявлением вы получите сообщение об ошибке.

Давайте подумаем, почему аргумент конструктора обязательно должен быть ссылкой? Представим, что ограничений на тип аргумента копирующего конструктора нет. В этом случае, когда main() вызовет функцию fn{}, компилятор C++ использует копирующий конструктор для создания копии объекта класса Student. При этом копирующий конструктор, получая объект по значению, требует вызова копирующего конструктора для создания копии объекта класса Student. При этом копирующий конструктор, получая объект по значению, требует вызова копирующего конструктора для создания копии объекта класса student. При этом копирующий конструктор, получая объект по значению, требует вызова копирующего конструктора для создания копии объекта класса student... И так до полного исчерпания памяти и аварийного останова.

Глава 20

Статические члены

В этой главе...

- ✓ Определение статических членов
- ✓ Объявление статических функций-членов
- ✓ Бюджет с классами — BUDGET2.CPP

По умолчанию данные-члены создаются отдельно для каждого объекта. Например, каждый студент имеет свое собственное имя.

Однако, кроме того, вы можете создавать данные-члены, используемые всеми объектами класса совместно, объявив их статическими. Несмотря на то что термин *статический* применим как к данным-членам, так и к функциям-членам, для данных и для функций его значение несколько различно. В этой главе рассматриваются оба типа и их отличия.

Определение статических членов

Данные-члены можно сделать общими для всех объектов класса, объявив их *статическими (static)*. Такие члены называются *статическими данными-членами* (я бы удивился, если бы они назывались по-другому...).

Зачем нужны статические данные

Большинство свойств класса являются свойствами отдельных объектов. Если использовать избитый (точнее, очень избитый) пример со студентами, можно сказать, что такие свойства, как имя, идентификационный номер и пройденные курсы, являются специфическими для каждого отдельного студента. Однако есть свойства, которые распространяются на всех студентов, например количество зачисленных студентов, самый высокий балл среди всех студентов или указатель на первого студента в связанном списке.

Такую информацию можно хранить в обшей (и ставшей привычной) глобальной переменной. Например, можно использовать простую целочисленную переменную для отслеживания количества объектов student. Однако при таком подходе возникает проблема, связанная с тем, что эти переменные находятся "снаружи" класса. Это подобно, например, установке регулятора напряжения моей микроволновой печи где-нибудь в спальне. Конечно, так можно сделать, и печь, вероятно, даже будет нормально работать, но моя собака во время очередной пробежки по квартире может наступить на провода и ее придется соскребать с потолка, что вряд ли доставит мне удовольствие (собаке, я думаю, это тоже не очень-то понравится).

Если уж создавать класс, полностью отвечающий за свое состояние, то такие глобальные переменные, как регулятор напряжения, нужно хранить внутри класса, подальше от неаккуратных собачьих лап. Это и объясняет необходимость статических членов.

Вы могли слышать, что статические члены также называют *членами класса,* поскольку они доступны для всех объектов класса. Чтобы подчеркнуть отличие от статических членов, обычные члены называют *компонентами экземпляра* или *членами объекта,* поскольку каждый объект имеет собственный набор таких членов.

Использование статических членов

Статические данные-члены объявляются в классе с помощью ключевого слова static, как показано в приведенном ниже примере.

```
class Student
{
public:
    Student fchar *pName = "no name")
    {
        strcpy(name, pName);
        noOfStudents++;
    }

    ~Student()
    {
        noOfStudents--;
    |
    int number()
    {
        return noOfStudents;
    }
protected:
    static int noOfStudents;
    char name[40];
};
Student s1;
Student s2;
```

Член noOfStudents входит в состав класса student, но не входит в состав объектов s1 и s2. Таким образом, для любого объекта класса Student существуют отдельные члены name и только один noOfStudents, который доступен для всех объектов класса Student.

"Хорошо, — спросите вы, — если место под noOfStudents не выделено ни в каком объекте класса Student, то где же он находится?" Ответ прост: это место не выделяется. Вы должны сами выделить для него место так, как показано ниже.

```
int Student::noOfStudents = 0;
```

Этот своеобразный синтаксис выщеляет место для статического члена класса и инициализирует его нулем. Статические данные-члены должны быть глобальными (как статические переменные не могут быть локальными по отношению к некоторой функции).



Для любого члена, имя которого встречается вне класса, требуется указание класса, к которому он принадлежит.

Обращение к статическим данным-членам

Правила обращения к статическим данным-членам те же, что и к обычным членам. Из класса к статическим членам обращаются так же, как и к другим членам класса. К открытым статическим членам можно обращаться извне класса, а к защищенным — нельзя, как и к обычным защищенным членам.

```
class Student
{
public:
    Student()
    {
```

```
        noOfStudents++; //обращение из класса
        /'/...остальнаяпрограмма...
    }

    static int noOfStudents;
    //...то же, что и раньше...
};
void fn(Student& s1, Student s2)
{
    //обращение к открытому статическому члену
    cout « "Количество студентов - "
        << s1.noOfStudents          //обращение извне
        << "\п";                    //класса
}
```

В функции fn () происходит обращение к noOfStudents с использованием объекта s1. Однако, поскольку s1 и s2 имеют одинаковый доступ к члену noOfStudents, возникает вопрос: почему я выбрал именно s1? Почему я не использовал s2? На самом деле это не имеет значения. Вы можете обращаться к статическим членам, используя любой объект класса, например, так:

```
//...класс определяется так же, как и раньше...
void fn:Student& s1, Student s2)
1
    //представленные команды приведут к идентичному результату
    cout << "Количество студентов - "
        << s1.noOfStudents <<"\n";
    cout << "Количество студентов - "
        << s2.noOfStudents << "\n";
}
```

На самом деле нам вообще не нужен объект! Можно использовать просто имя класса, как показано в следующем примере:

```
//...класс определяется так же, как и раньше...
void fn;student& s1, Student s2)
{
    // результат остается неизменным
    cout << "Количество студентов - "
        << Student::noOfStudents
        << "\n";
}
```

Независимо от того, будете ли вы использовать имя объекта или нет, C++ все равно будет использовать имя класса.



Объект, используемый для обращения к статическому члену, никак не обрабатывается, даже если это обращение явным образом указано в выражении. Для того чтобы понять, что я имею в виду, рассмотрим приведенный ниже пример.

```
class Student
{
    static int noOfStudents;
    Student& nextStudent ();
    //...то же, что и раньше...
};
void fn(Student& s)
{
    cout « s.nextStudent().noOfStudents << "\n";
}
```

Функция-член nextStudent(> в этом примере не вызывается. Все, что нужно знать С++ для обращения к noOfStudents, — это тип возвращаемого значения, а он может это выяснить и не выполняя эту функцию. Хотя этот пример и маловразумите" лен, но так оно и будет[16].

Применение статических данных-членов

Существует бесчисленное множество областей применения статических данных-членов, но здесь мы остановимся лишь на нескольких из них. Во-первых, можно использовать статические члены для хранения количества объектов, задействованных в программе. Например, в классе Student такой счетчик можно проинициализировать нулем, а затем увеличивать его на единицу внутри конструктора и уменьшать внутри деструктора. Тогда в любой момент этот статический член будет содержать количество существующих в данный момент объектов класса Student. Однако следует помнить, что этот счетчик будет содержать количество объектов, существующих вданный момент (включая временные объекты), а не количество студентов[17].

Еще один способ использования статических членов заключается в индицировании выполнения определенного действия. Например, классу Radio может понадобиться инициализировать некие аппаратные средства при первом выполнении команды tune, но не перед последующими вызовами. С помощью статического члена можно указать, что первый вызов tune уже выполнил инициализацию. Кроме всего прочего, статический член может служить указателем безошибочности инициализации аппаратных средств.

И наконец, в статических членах можно хранить указатель на первый элемент связанного списка. Таким образом, статические члены могут содержать любую информацию "общего использования", которая будет доступна для всех объектов и во всех функциях (не стоит забывать, однако, что чрезмерное использование статических переменных усложняет поиск ошибок в программе).

Объявление статических функций-членов

Функции-члены также могут быть объявлены статическими. Подобно статическим данным-членам, они связаны с классом, а не с каким-либо отдельным объектом класса. Это означает, что обращение к статическим функциям-членам, как и к статическим данным-членам, не требует наличия объекта. Если объект и присутствует, то используется только его тип.

Таким образом, оба вызова статической функции-члена number O в приведенном ниже примере легальны.

```
#include <iostream.h>
#include <string.h>
```

---

16 *Вообще говоря, это зависит от используемого компилятора. Так, GNU С++ не будет вызывать функцию, в то время как Borland С++ или Watcom С++ сделает это.* — Прим. ред.

17 *Еще одно замечание: в этом случае вы должны позаботиться о том, чтобы счетчик увеличивался во всех конструкторах, включая конструктор копирования.* — Прим. ред.

```
class Student
{
public:
    static int number()
    f
        return noOfStudents;
    }

    //. ..все остальное то же, что и ранее...
protected:
    static int noOfStudents;
    char name[40];
};
int Student::noOfStudents = 0;
int main(int argcs, char* pArgs[])
{
    Student s;
    cou- << s.number() << "\n";
    cout << Student::number() << "\n";
    return 0;
}
```

Обратите внимание на то, как статическая функция-член обращается к статическим данным-членам. Поскольку статическая функция-член не связана с каким-либо объектом, она не может неявно обращаться к нестатическому члену. Таким образом, приведенный ниже пример неправилен.

```
class Student
{
public:
    //приведенный ниже код неверен
    static char *sName()
    {
        return name;     //Какое именно имя?
    }

    //...все остальное то же, что и ранее...
protected:
    char name[40];
    static int noOfStudents;
};
```

Это не означает, что статические функции-члены не имеют доступа к нестатическим данным-членам. Рассмотрим следующий пример:

```
#include <iostream.h>
#include <string.h>
class Student
(
public:
    //те же конструктор и деструктор, что и ранее,
    //однако конструктор обеспечивает вставку объекта
    //в связанный список (как это делается, сейчас
    //для нас неважно...
    Student (char *pName);
    ~Student () ;
    //findName - возвращает студента с указанным именем
    static Student* findName(char *pName);
protected:
```

```
    static Student *pFirst;
    Student *pNext;
    char name[40];
};
Student* Student::pFirst = 0;

//findName — возвращает студента
//           с указанным именем
//           либо ноль, если такой
//           студент не был найден.
Student* Student::findName(char *pName)
f
    //просмотрим связанный список...
    for (Student *pS = pFirst; pS; pS = pS->pNext)
    {
        //если указанное имя найдено...
        if (strcmp(pS->name, pName) == 0)
        {
            //...то возвращаем адрес этого объекта
            return pS;
        }
    }
    //в противном случае возвращаем ноль(объект не найден)
    return (Student*)0;
}
int main(int argcs, char* pArgs[])
{
    Student s1("Randy");
    Student s2("Jenny");
    Student s3("Kinsey");
    Student *pS = Student::findName("Jenny");
    return 0;
}
```

Функция findName() имеет доступ к pFirst, поскольку этот указатель доступен для всех объектов. Так как findName является членом класса student, он имеет доступ к членам name объектов, однако при вызове необходимо уточнить, член какого именно объекта требуется. Статическая функция не связана с каким-либо определенным объектом, и делу не поможет даже использование объекта при вызове статического члена.

```
int main(int argcs, char* pArgs[])
{
    Student s1("Randy");
    Student s2("Jenny");
    Student s3("Kinsey");
    Student *pS = s1.findName("Jenny");
    return 0;
}
```

Объект s1 не передается функции findName O; компилятор использует только класс этого объекта, для того чтобы знать, функцию findName () какого именно класса следует вызвать.

Статические функции-члены удобны, когда вам надо связать некое действие с классом, не связывая его с отдельным объектом. Например, функция-член Duck::fly() (Утка::полет()) ассоциируется с определенной уткой, тогда как более "трагичная" функция Duck::goExtinct(> (Утка{)::Вымирание ()) связана со всем утиным племенем (читай: классом).



Что такое this

Я уже упоминал несколько раз о том, что такое this, тем не менее давайте еще раз разберемся в этом вопросе. this — это указатель на текущий объект внутри функции-члена. Он используется, когда не указано другое имя объекта. В обычной функции-члене this — скрытый первый аргумент, передаваемый функции.

```
class SC
{
public:
    void nFn(int a);
        //то же, что и SC::nFn(SC *this, int a)
    static void sFn(int a);
        //то же, что и SC::sFn(int a)
};

void fn(SC& s)
f
  s.nFn(lO);   //Преобразуется в SC::nFn(ss, 10);
  s.sFn(lO);   // Преобразуется в SC::sFn(10);
}
```

Таким образом, функция nFn () интерпретируется так же, как если бы мы объявили ее void SC::nFn(SC *this, int a). При вызове nFn() неявным первым аргументом ей передается адрес s (вы не можете записать вызов таким образом, так как передача адреса объекта — дело компилятора).

Обращения к другим, не статическим членам из функции SC: :sFn автоматически используют аргумент this как указатель на текущий объект. Однако при вызове статической функции SC: : sFn () адрес объекта ей не передается и указателя this, который можно использовать при обращении к нестатическим членам, не существует. Поэтому мы и говорим, что статическая функция-член не связана с каким-либо текущим объектом.

Бюджет с классами - BUDGET2 . CPP

В этом разделе обсуждается новая версия программы BUDGET, с которой вы уже встречались во второй части книги. Однако, в отличие от предыдущей "функциональной" версии, эта программа будет объектно-ориентированным решением, основанным на классах.

В программе будет решаться задача по представлению счетов в том виде, какой они имеют в банке. Эта простая программа будет поддерживать возможности вклада (это хорошо) и снятия денег со счета (это еще лучше). (Первоначальная версия бюджета, приведенная в конце предыдущей части, работала только с одним типом банковского счета.) Эта версия поддерживает два типа счетов, каждый из которых будет иметь собственные, несколько отличные от другого счета правила.

Чековый счет:

- ✓ удерживать 20 центов за каждый обработанный чек, если баланс падает ниже 500 долларов;
- ✓ не удерживать 20 центов, если баланс больше 500 долларов.

Сберегательный счет:

- ✓ не удерживать денег при первом снятии со счета за месяц;
- ✓ удерживать 5.00 доллара за каждое последующее снятие.

Рассматривая эту задачу, можно сразу отметить, что главными кандидатами на роль классов являются Checking и Savings. Поскольку данные-члены лучше сделать защищенными, нам понадобится несколько функций, обеспечивающих доступ к номеру и балансу счета.

Как и любой класс, Checking и Savings нуждаются в конструкторе, чтобы проинициализировать объекты правильными значениями (как минимум, обнулить баланс). Кроме того, понадобятся еще две функции — deposit () (вклад) и withdrawal () (снятие).

И наконец, в этой программе я добавил еще одну функцию-член, которая называется display (); она занимается отображением текущего объекта. Это необязательное требование, однако обычно так и поступают, позволяя объекту самому заниматься своим отображением, не полагаясь на внешнюю функцию (которой для правильного отображения может понадобиться информация о внутреннем устройстве класса или другая информация, которую вы, возможно, не захотите открывать).

Вот текст этой программы.

```
// BUDGET2.CPP — программа бюджета, основанная на классах.

#include <iostream.h>
#include <stdio.h>

//максимальное количество счетов
const int maxAccounts = 10;
// Checking — здесь описан чековый счет
class Checking
{
public:
    Checking(int initializeAN = 0)
    {
        accountNumber = initializeAN;
        balance = 0.0;
    }

// функции обращения
    int accountNo()
    {
        return accountNumber;
    }
    double acntBalance()
    {
        return balance;
    )
    //функции транзакций
    void deposit(double amount)
    {
        balance += amount;
    )
    void withdrawal(double amount);

    // функция вывода объекта в cout
    void display()
    {
        cout << "Счет " << accountNumber
```

```
             << " = "   << balance
             << "\n";
    }

protected:
    unsigned accountNumber;
    double  balance;
};
// withdrawal — эта функция-член слишком велика
//              для inline-функции
void Checking::withdrawal(double amount)
{
    if (balance < amount)
    {
        cout << "Недостаточно денег: баланс равен "
             << balance
             << ", сумма чека равна " << amount
             << "\n";
    }
    else
    {
        balance -= amount;
        // если баланс падает слишком низко...
        if (balance < 500.00)
        {
        // ...удержать деньги за обслуживание
            balance -= 0.20;
        }
    }
}

// Savings — вы и сами можете написать этот класс
class Savings
{
public:
    Savings(int initialAN = 0)
    {
        accountNumber = initialAN;
        balance = 0.0;
        noWithdrawals = 0;
    }

    // функции обращения
    int accountNo()
    {
        return accountNumber;
    }
    double acntBalance()
    {
        return balance;
    }

    // функции транзакций

    void deposit(double amount)

    }   balance += amount;

    void withdrawal(double amount);
```

```
    // функция display — отображает объект на 'cout'
    void display()
    {
        cout << "Счет "  << accountNumber
             << " = " << balance
             << "  (номер снятия = "
             «  noWithdrawals
             << ")\n";
    }
protected:
    unsigned accountNumber;
    double   balance;
    int      noWithdrawals;
};
void Savings::withdrawal(double amount)
{
    if  (balance < amount}
    {
        cout << "Недостаточно денег на счете:  "
             << "баланс равен " << balance
             << ", снимается " << amount
             << "\n";
    }
    else
    {
    // после первого в месяце снятия денег...
        if  (++noWithdrawals > 1)
        {
        //  ...удерживать S5
            balance -= 5.00;
        }

        // снять деньги
        balance -= amount;
    }
}

// объявление прототипов
void process(Checking* pChecking);
void process(Savings*  pSavings);

// объекты чековых и сберегательных счетов
Checking* chkAcnts[maxAccounts];
Savings*  svgAcnts[maxAccounts];

// main — собирает и выводит данные
int main(int argcs, char* pArgs[])
{
    // повторять цикл до ввода 'X' или 'x'
    int noChkAccounts = 0;  // содержит количество счетов
    int noSvgAccounts = 0;
    char accountType;   // тип счета — 'S' или 'C'
    while  (1)
    {
        cout << "Введите S для сберегательных счетов,  "
             << "C для чековых,  "
             << "X для выхода:";
        cin >> accountType;

    // выйти из цикла, если пользователь введет X
```

```
if (accountType == 'x' || accountType == 'X')
i
    break;;
}

// E противном случае обрабатывать соответствующий счет
switch (accountType)
{
// чековые счета
    case 'c':
    case 'C':
    if {noChkAccounts < maxAccounts)
    {
        int acnt;
        cout << "Введите номер счета:";
        cin  >> acnt;
        chkAcnts[noChkAccounts] = new Checking(acnt);
        process(chkAcnts[noChkAccounts]);
        noChkAccounts++;
    }
    else
    {
        cout << "Для чековых счетов больше нет места\n";
    }
    break;

    // сберегательные счета
    case 's':
    case 'S':
    if (noSvgAccounts < maxAccounts)
    {
        int acnt;
        cout << "Введите номер счета:";
        cin  >> acnt;
        svgAcnts[noSvgAccounts] = new Savings(acnt);
        process(svgAcnts[noSvgAccounts]);
        noSvgAccounts++;
    }
    else
    {
        cout << "Для сберегательных счетов больше нет места\n";
    }
    break;

    default:
        cout << "Непонятный символ...\п";
    }
}

//А теперь показать общую сумму
double chkTotal = 0;
cout << "Чековые счета:\n";
for (int i = 0; i < noChkAccounts; i++)
{
    chkAcnts[i]->display();
    chkTotal += chkAcnts[i]->acntBalance();
}

double svgTotal = 0;
cout << "Сберегательные счета:\n";
```

```
    for (int j = 0 ; j < noSvgAccounts; j++)
    i
        svgAcnts[j]->display();
        svgTotal += svgAcnts[j]->acntBalance();
    }

    double total = chkTotal + svgTotal;
    cout << "Сумма по чековым счетам = "
         « chkTotal
         << "\n";

    cout << "Сумма по сберегательным счетам = "
         << svgTotal
         << "\n";

    cout << "Общая сумма          = "
         << total
         << "\n";
    return 0;
}

// обработка(Checking) — ввод данных по чековым счетам
void process (Checking* pCheckir.g)
{
    cout << "Введите положительное число для вклада,\п"
         << "отрицательное для снятия, 0 для завершения\п";
    double transaction;
    do
    {
        cout << " : ";
        cin >> transaction;

        // вклад
        if (transaction > 0)
        {
            pChecking->deposit(transaction);
        }

        // снятие
        if (transaction < 0)
        {
            pChecking->withdrawal(-transaction);
        }
    ) while (transaction J = 0);
}

// обработка(Savings) — ввод данных для сберегательных счетов
void process(Savings* pSavings)
{
    cout << "Введите положительное число для вклада,\n"
         << "отрицательное для снятия, 0 для завершения\n";
    double transaction;
    do
    {
        cout << ":";
        cin >> transaction;

        // вклад
        if (transaction > 0)
        {
```

```
            pSavings->deposit(transaction);
        }

        // снятие
        if (transaction < 0)
        {
            pSavings->withdrawal(-transaction);
        }
    } while (transaction != 0);
}
```

Я запустил эту программу с приведенными ниже данными для того, чтобы продемонстрировать, как она работает (хотя почему-то чаще мои программы не работают). **Жирным** шрифтом выделен пользовательский ввод, а обычным представлены сообщения программы.

```
Введите 8 для сберегательных счетов,
        С для чековых, X для выхода:S
Введите номер счета:123
Введите положительное число для вклада,
отрицательное для снятия, 0 для завершения
:200
:-20
:0
Введите S для сберегательных счетов,
        С для чековых, X для выхода:S
Введите номер счета:234
Введите положительное число для вклада,
отрицательное для снятия, 0 для завершения
:200
:-10
:-10
:0
Введите 3 для сберегательных счетов,
С для чековых, X для выхода:C
Введите номер счета:345
Введите положительное число для вклада,
отрицательное для снятия, 0 для завершения
:200
:-20
:0
Введите 3 для сберегательных счетов,
С для чековых, X для выхода:C
Введите номер счета:456
Введите положительное число для вклада,
отрицательное для снятия, 0 для завершения
:600
:-20
:0
Введите S для сберегательных счетов,
С для чековых, X для выхода:X
Чековые счета:
Счет 345 = 179.8
Счет 456 = 580
Сберегательные счета:
Счет 123 = 180 (номер снятия = 1)
Счет 234 = 175 (номер снятия = 2)
Сумма по чековым счетам = 7 59.8
```

```
Сумма по сберегательным счетам = 355
Общая сумма          = 1114.8
```

Рассмотрим каждую из функций-членов, начиная с класса Checking. Конструктор присваивает счету его номер. Значение по умолчанию "= 0" позволяет программе создавать объект с номером счета по умолчанию, равным нулю.

```
Checking c1 = new Checking(124);
Checking c2 = new Checking();
```

В данном случае объект c1 класса Checking создается с номером счета, равным 123, тогда как объект c2 создается с номером счета по умолчанию, который равен нулю.

Функции accountNo() и acntBalance () предоставляют внешнему миру доступ к защищенным членам accountNumber и balance. Задачей этих функций является предоставление внешним функциям — не членам значений, изменить которые невозможно. Кроме того, эти функции, обеспечивающие доступ к членам, предохраняют внешние функции от необходимости внесения изменений при переменах в методе хранения номера счета или баланса.

Функции deposit() и withdrawal() отвечают за вложение и снятие денег со счета. Поскольку функция deposit() довольно проста, она была определена как inline-функция. Функция withdrawal (), будучи несколько сложнее, объявлена в классе, но определяется позже.

Функция display () выводит важные данные на устройство стандартного вывода.

Класс Savings, в сущности, идентичен классу checking, за исключением дополнительного члена noWithdrawals, который используется для отслеживания количества проведенных снятий.

Место под объекты сберегательного и чекового счета выделяется в массивах svgAcnts и chkAcnts соответственно. Максимальное количество счетов определено величиной maxAccounts.

Функция main() несколько сложнее своей сестры из программы Budget1, поскольку она имеет дело с двумя разными типами счетов. После проверки ввода на равенство "X" функция main() использует конструкцию switch, чтобы выбрать тип счета: с для чекового и S для сберегательного. Конструкция switch использована в этой программе по двум причинам: во-первых, ее проще расширить, добавляя к ней дополнительные варианты; во-вторых, она предоставляет вариант default (вариант по умолчанию) для обработки неверного ввода.

Как и ранее, вторая часть функции main() обеспечивает отображение информации о счете, собранной в первой части этой функции.

Обратите внимание на то, как содержимое классов Checking и Savings скрыто от main (). Так, например, main () просит объект показать свое содержимое, однако при этом не имеет никакого представления о том, как класс выбирает, что именно и как это показать.

Функция process (), которая обрабатывает текущие вложения и снятия, полагается на функции-члены deposit () и withdrawal (), которые выполняют за нее всю черную работу. Хотя вы и знаете, как именно выполняются эти действия, помните, что process () об этом не имеет никакого понятия. Работа счета касается только самого класса счета.

Советую пройти эту программу в пошаговом режиме. Ничто иное не даст более полного представления о программе, чем ее рассмотрение в действии.

Хотите — верьте, хотите — нет, но с позиции программирования Budget2 разрабатывается легче, чем Budget1. Когда я писал класс Savings, я не должен был волноваться о том, как он будет использоваться главной программой (то же относится и к классу Checking). Когда же я работал над функцией main (), то не думал о содержимом класса.

Однако в этой программе есть один небольшой недостаток: классы Savings и Checking имеют очень много общего, и хотелось бы найти возможность уменьшить количество повторений кода. Эта возможность и является темой части 4, "Наследование", в конце которой вы найдете очередную версию нашей программы — Budget3.

Часть IV

Наследование



В этой части...

Из дискуссии по вопросам объектно-ориентированной философии в части 3 становится ясно, что в реальном мире существует две вещи, которые нельзя выразить с помощью функционально-ориентированных программ. Первое — это возможность работы с отдельными объектами. Я привел пример использования микроволновой печи для приготовления закуски. Она предоставляет интерфейс (на лицевой панели), который я использую для управления, совершенно не вникая в подробности работы печи. Я буду вести себя точно так же, даже если буду знать все о том, как именно она устроена (хотя я этого не знаю). Второй аспект реального мира, закрытый для функциональных программ, — это классификация объектов: распознавание и использование их подобия. Если в рецепте приготовления того или иного блюда указана печь любого типа, то, работая с микроволновой печью, я буду уверен, что использую правильное устройство, поскольку микроволновая печь является одним из типов печей. В предыдущей части вы познакомились с механизмом, используемым в C++ для осуществления первой возможности объектно-ориентированного программирования, — с классами. Для обеспечения второй возможности C++ использует концепцию, называемую наследованием, *которая расширяет понятие и возможности классов. Именно о наследовании и пойдет речь в этой части книги.*

Глава 21

Наследование классов

В этой главе...

- ✓ Зачем нужно наследование
- ✓ Как наследуется класс
- ✓ Конструирование подкласса
- ✓ Отношение СОДЕРЖИТ

В этой главе обсуждается *наследование* (inheritance), т.е. способность одного класса наследовать возможности или свойства другого класса.

Наследование — это общепринятая концепция. Я — человек (за исключением раннего утра...). И я наследую некоторые свойства класса Человек, например возможность говорить (в большей или меньшей степени), интеллект (надеюсь, что в большей степени), необходимость в воздухе, воде, пище и всяких витаминах. Эти свойства не являются уникальными для каждого отдельного человека. Очевидно, что класс Человек наследует зависимость от воды, воздуха и пищи у класса Млекопитающие, который, в свою очередь, наследует эти свойства у класса Животные.

Концепция, в основе которой лежит способность передавать свойства по наследству, очень мошная. Благодаря ей можно значительно сэкономить место при описании реального объекта. Например, если мой сын спросит: "Что такое утка?", я смогу сказать: "Это птица, которая крякает". Несмотря на краткость, этот ответ несет в себе всю необходимую для описания утки (по крайней мере, для моего сына) информацию. Мой сын знает, что такое птица, и может понять, что утке присущи все свойства птицы плюс свойство "кряканье".

В объектно-ориентированных языках такая наследственная связь выражается в возможности одного класса наследовать другой. Таким образом, объектно-ориентированные языки позволяют создавать модели, более близкие к реальному миру (а именно для этого они и созданы), чем модели, построенные с помощью языков, не поддерживающих наследование.

В С++ один класс может наследовать другой следующим образом:

```
class Student
{
};
class GraduateStudent : public Student
{
};
```

В этом примере GraduateStudent наследует все члены класса Student. Таким образом, GraduateStudent ЯВЛЯЕТСЯ студентом. Конечно, при этом GraduateStudent может также содержать уникальные, присущие именно ему члены.

Зачем нужно наследование

Наследование было включено в С++ по нескольким причинам. Конечно, основной из них была необходимость выражать связи между классами с помощью наследования (к этому я еще вернусь). Менее важной целью было уменьшение размера исходного

кода. Представьте себе, что у вас есть класс Student и вас попросили добавить новый класс под названием GraduateStudent. В этом случае наследование значительно уменьшит количество членов, которые вам придется добавлять в класс. Все, что вам действительно нужно в классе GraduateStudent, — это члены, которые будут описывать отличия между студентами и аспирантами.

С этим связана и более важная проблема — необходимость механизма повторного использования. Разработчики программного обеспечения на каком-то этапе поняли, что глупо начинать каждый новый проект с нуля, заново создавая одни и те же программные компоненты.

Сравните ситуацию в области создания программных продуктов с другими сферами производства. Много ли вы знаете автомобилестроителей, которые начинают конструировать новый автомобиль с создания новых гаек и винтов? Даже если отдельные компании так и делают, вряд ли будет легко отыскать среди них такие, которые начинают с разработки отверток и гаечных ключей. Разработчики из других областей давно знают, что гораздо эффективнее создавать новую машину из уже существующих гаек, винтов, креплений и даже более сложных агрегатов — компрессоров и двигателей.

К сожалению, такая простая философия слишком мало практикуется в области программного обеспечения. За исключением ряда мелких функций, входящих в состав стандартных библиотек С, редко встретишь повторное использование программных компонентов. Одной из проблем, связанных с повторным использованием, является почти полная невозможность отыскать в готовой программе компонент, который бы отвечал всем вашим требованиям. Обычно все уже существующие компоненты приходится дописывать и переписывать перед применением в данном приложении (как если бы работа по созданию самолета состояла в том, чтобы убрать с помощью напильника все лишнее у паровоза...).

В программировании есть правило: "Если ты что-то открыл, значит, ты это уже сломал". Другими словами, если вам пришлось переписать функцию или класс, чтобы приспособить ее к новому приложению, то вам придется заново протестировать всю функцию, а не только те части, которые вы изменили. Изменения могут внести ошибки в любое место существующего кода.

Наследование позволяет приспосабливать существующие классы к новым приложениям, не внося изменений в их внутреннее устройство. Существующий класс наследуется новым подклассом, который и будет содержать все необходимые добавления и изменения.

Это приводит нас к третьему преимуществу наследования. Представьте себе, что вы наследуете некий существующий класс. Затем вы находите, что базовый класс содержит ошибку, которую нужно исправить. Если бы вы переделывали класс для его повторного использования, вам бы пришлось вручную проверить новый класс, отыскивая ошибку в каждом отдельно взятом приложении. Однако если вы наследовали класс без изменений, то можете без особых хлопот заменить старую версию класса новой.

Это потрясающе

Люди составляют обширные системы, чтобы было проще разбираться в том, что их окружает. Тузик является частным случаем собаки, которая является частным случаем собакообразных, которые входят в состав млекопитающих, и т.д. Так легче познавать мир.

Если использовать другой пример, можно сказать, что студент является человеком (точнее, его частным случаем). Как только это сказано, я уже знаю довольно много о студентах (об американских студентах, естественно). Я знаю, что они имеют номера социального страхования, что они слишком много смотрят телевизор и постоянно мечтают о сексе. Я знаю все это потому, что это свойства всех людей.

В C++ мы говорим, что класс Student наследует класс Person. Кроме того, мы говорим, что Person является *базовым классом* для класса student. Наконец, мы говорим, что student ЯВЛЯЕТСЯ Person (использование прописных букв — общепринятый метод отражения уникального типа связи; не я это придумал). Эта терминология используется в C++ и других объектно-ориентированных языках программирования.

Заметьте, что хотя Student и ЯВЛЯЕТСЯ Person, обратное не верно. Person не ЯВЛЯЕТСЯ Student (такое выражение следует трактовать в общем смысле, поскольку конкретный человек, конечно же, может оказаться студентом). Существует много людей, которые являются членами класса Person и не являются членами класса Student. Кроме того, класс student имеет средний балл, а Person его не имеет.

Свойство наследования транзитивно. Например, если я определю новый класс GraduateStudent как подкласс класса Student, то он тоже будет наследником Person. Это значит, что будет выполняться следующее: если GraduateStudent ЯВЛЯЕТСЯ Student и Student ЯВЛЯЕТСЯ Person, то GraduateStudent ЯВЛЯЕТСЯ Person.

Как наследуется класс

Здесь приведен пример уже рассмотренного класса GraduateStudent, который дополнен несколькими членами.

```
#include <string.h>
class Advisor
{
};

class Student
{
public:
    Student(char *pName = "no name")
    {
        strncpy(name, pName, sizeof(name));
        average = 0.0;
        semesterHours = 0;
    }
    void addCourse(int hours, float grade)
    {
        average = (semesterHours * average + grade);
        semesterHours += hours;
        average = average / semesterHours;
    }

    int hours( ) { return semesterHours;}
    float gpa( ) { return average;}
protected:
    char name[40];
    int semesterHours;
    float average;
};
class GraduateStudent : public Student
```

```
{
public:
    int qualifier( > { return qualifierGrade;};

protected:
    Advisor advisor;
    int qualifierGrade;
};

int main( )
{
    Student llu("Lo Lee Undergrad");
    GraduateStudent gs;
    llu.addCourse(3, 2.5);
    gs.addCourse(3, 3.0);
    return 0;
}
```

В этом примере класс Student содержит тс же члены, что и ранее. Объект llu — просто один из объектов Student. Класс GraduateStudent несколько отличается по структуре от класса Student; двоеточие и следующее за ним public Student объявляет класс GraduateStudent наследником Student.

Ключевое слово public наводит на мысль о том, что может существовать и *защищенное* наследование, однако его обсуждение выходит за рамки нашей книги (вы еще не забыли этот термин?).

Объект gs как член подкласса Student может делать то же, что и объект llu. Он содержит данные-члены name, semesterHours, average и функцию-член addCourse (). Кроме того, GraduateStuder.t содержит также члены qualifier(), advisor и qualifierGrade- Таким образом, gs в прямом смысле этого слова ЯВЛЯЕТСЯ классом Student плюс кое-что еше.

Разберем приведенный ниже сценарий.

```
void fn(Student& s)
{
    // ...все, что угодно...
}

int main( )
{
    GraduateStudent gs;
    fn(gs) ;
    return 0;
}
```

Обратите внимание, что функция fn() принимает в качестве аргумента ссылку на объект класса student. Однако в main() мы передаем ей объект класса GraduateStudent. Функция спокойно принимает этот аргумент именно потому, что (а ну-ка еше раз, все вместе!) "GraduateStudent ЯВЛЯЕТСЯ Student".

Это же условие позволяет вызывать функцию-член класса Student из объекта класса GraduateStudent, как это делается в приведенном ниже примере.

```
int main( )
[
    GraduateStudent gs;
    //вызываем функцию Student::addCourse()
    gs.addCourse(3, 2.5) ;
    return 0;
}
```

Конструирование подкласса

Хотя подкласс и имеет доступ к защищенным членам базового класса, а значит, может инициализировать их, было бы хорошо, если бы базовый класс все же конструировал сам себя. В действительности так и происходит.

Перед тем как управление получает код, стоящий за открывающей фигурной скобкой класса GraduateStudent, оно передается конструктору по умолчанию класса Student (поскольку другой конструктор не был указан). Если бы класс student был наследником другого класса, например Person, то конструктор этого класса вызывался бы перед передачей управления конструктору student. Подобно небоскребу, объект строится, начиная с "фундаментального" уровня в соответствии со структурой наследования классов и вызывая конструкторы всех классов, составляющих данный.

Как и в случае с объектами-членами, вам может понадобиться передавать аргументы конструктору базового класса. Это делается почти так же, как и изученная ранее передача аргументов конструктору объекта-члена (смотрите приведенный ниже пример).

```
class GraduateStudent : public Student
{
public:
    GraduateStudent(char *pName,
                    Advisors adv) : Student(pName),
                                    advisor(adv)
    {
        qualifierGrade = 0;
    }

//все остальное такое же, как и раньше
};

void fn(Advisor& advisor)
{
    GraduateStudent gs("Yen Kay Doodle", advisor);
    // ...все остальное, что должна делать эта функция...
}
```

ВЭТОМпримере конструктор класса GraduateStudent вызывает конструктор Student, передавая ему аргумент pName. Базовый класс конструируется до любых объектов-членов, а значит, конструктор класса Student вызывается перед конструктором Advisor. И только после конструктора Advisor (который вызывается для члена advisor) начинает работу конструктор GraduateStudent.

Следуя правилу о том, что деструкторы вызываются в порядке, обратном вызову конструкторов, первым вызывается деструктор GraduateStudent. После того как он выполнит свою работу, управление передается деструктору класса Advisor, а затем деструктору Student. Если бы student был наследником класса Person, его деструктор получил бы управление после деструктора Student.

И это логично. Блок памяти сначала преобразуется в объект student, а уже затем конструктор для GraduateStudent превращает этого студента в аспиранта. Деструктор же просто выполняет этот процесс в обратном направлении.

Отношение СОДЕРЖИТ

Обратите внимание, что класс GraduateStudent включает в себя члены классов Student и Advisor, однако он включает их по-разному. Определяя данные-члены класса Advisor, вы знаете, что класс Student содержит внутри все данные-члены класса Advisor, но вы не можете сказать, что GraduateStudent ЯВЛЯЕТСЯ Advi-

sor. Однако вы можете сказать, что GraduateStudent СОДЕРЖИТ Advisor. Какая разница между этим отношением и наследованием?

Используем в качестве примера автомобиль. Вы можете логически определить автомобиль, как подкласс транспортных средств, а значит, он будет наследовать свойства остальных транспортных средств. С другой стороны, автомобиль содержит мотор. Если вы покупаете автомобиль, то покупаете и мотор (если, конечно, вы не покупаете бывшую в употреблении машину там же, где я купил свою кучу металлолома).

Если друзья пригласят вас приехать на воскресный пикник на новой машине и вы приедете на ней, никто не будет удивлен (даже если вы явитесь на мотоцикле), поскольку автомобиль ЯВЛЯЕТСЯ транспортным средством. Но если вы появитесь на своих двоих, неся в руках мотор, друзья решат, что вы попросту издеваетесь над ними, поскольку мотор не является транспортным средством, так как не имеет некоторых важных свойств, присущих транспортным средствам.

В аспекте программирования связь типа СОДЕРЖИТ достаточно очевидна. Разберем следующий пример:

```
class Vehicle
{
};
class Motor
{
};
class Car : public Vehicle
{
public:
    Motor motor;
};

void VehicleFn(Vehicle& v);
void motorFn(Motor& m);

int main()
{
    Car c;
    VehicleFn(c);        //так можно вызвать
    motorFn(c);          //а так — нельзя
    motorFn(c.motor);    //нужно вот так
    return 0;
}
```

Вызов VehicleFn(c) допустим, поскольку с ЯВЛЯЕТСЯ Vehicle. Вызов motorFn(c) недопустим, поскольку с — не Motor, хотя он и содержит Motor. Если возникает необходимость передать функции только ту часть с, которая является мотором, это следует выразить явно: motorFn(c.motor).

Глава 22

Знакомство с виртуальными функциями-членами: настоящие ли они

В этой главе...

- ✓ Зачем нужен полиморфизм
- ✓ Как работает полиморфизм
- ✓ Полиморфное приготовление закуски
- ✓ Когда функция не является виртуальной
- ✓ Виртуальные особенности

Количество и тип аргументов функции включены в ее полное или, другими словами, расширенное имя. Это позволяет создавать в одной программе функции с одним и тем же именем (если различаются их полные имена):

```
void someFn(int)
void someFn (char*)
void someFn (char* , double)
```

Во всех трех случаях функции имеют одинаковое короткое имя someFn(). Полные имена всех трех функций различаются: someFn(int) отличается от someFn(char*) и т.д. C++ решает, какую именно функцию нужно вызвать, рассматривая полные имена слева направо.



Тип возвращаемого значения не является частью полного имени функции, поэтому вы не можете иметь две функции с одинаковым расширенным именем, отличающиеся только типом возвращаемого объекта.

Итак, функции-члены могут быть перегружены. При этом помимо количества и типов аргументов расширенное имя функции-члена содержит еше и имя класса.

С появлением наследования возникает небольшая неувязка. Что, если функция-член базового класса имеет то же имя, что и функция-член подкласса? Попробуем разобраться с простым фрагментом кода:

```
class Student
(
public:
    //...то же, что и раньше...
    float calcTuition();
};

class GraduateStudent : public Student
{
public:
    float calcTuition();
};
```

```
int main (int argcs, char* pArgs[])
{
    Student s;
    GraduateStudent gs;
    s.calcTuition();    //вызывает Student::calcTuition()
    gs.calcTuition();  //вызывает...
                        //GraduateStudent::calcTuition()
    return 0;
}
```

Как и в любой ситуации с перегрузкой, когда программист обращается к calcTuition(), C++ должен решить, какая именно функция calcTuition() вызывается. Если две функции отличаются типами аргументов, то нет никаких проблем. Даже если аргументы одинаковы, различий в именах класса достаточно, чтобы решить, какой именно вызов нужно осуществить, а значит, в этом примере нет ничего необычного. Вызов s.calcTuition() обращается к Student::calcTuition(), поскольку s локально объявлена как Student, тогда как gs.calcTuition() обращается к GraduateStudent::calcTuition().

Но что, если класс объекта не может быть точно определен на этапе компиляции? Чтобы продемонстрировать подобную ситуацию, нужно просто немного изменить приведенную выше программу:

```
class Student
{
public:
    //то же, что и раньше
    float calcTuition()
    {
        return 0;
    }
};

class GraduateStudent : public Student
{
public:
    float calcTuition()
    {
        return 0;
    }
};

void fn(Student& x)
{
    x.calcTuition();  //к какому из calcTuition()
                      //относится эта строка?
}
int main(int argcs, char* pArgs[])
{
    Student s;
    GraduateStudent gs;
    fn(s);
    fn(gs);
    return 0;
}
```

На этот раз вместо прямого вызова calcTuition() осуществляется вызов через промежуточную функцию fn(). Теперь все зависит от того, какой аргумент передается fn(), поскольку x может быть как Student, так и Graduate-Student. Ведь GraduateSudent ЯВЛЯЕТСЯ Student!



Если вы этого не знали, это вовсе не говорит о том, что вы ЯВЛЯЕТЕСЬ чайником. Это значит, что вы не читали главу 21, "Наследование классов".

Аргумент x, передаваемый fn(). для экономии места и времени объявлен как ссылка на объект класса Student. Если бы этот аргумент передавался по значению, C++ пришлось бы при каждом вызове fn() конструировать новый объект Student. В зависимости от вида класса Studer.t и количества вызовов fn() в итоге это может занять много времени, тогда как при вызове fn(Students) или fn(student*) передается только адрес. Если вы не поняли, о чем я говорю, перечитайте главу 15, "Создание указателей на объекты".

Было бы неплохо, если бы строка x.calcTuition{> вызывала Student::calcTuition(), когда x является объектом класса Student, и GraduateSudent::calcTuition(), когда x является объектом класса GraduateStudent. Если бы C++ был настолько "сообразителен", это было бы действительно здорово! Почему? Об этом вы узнаете далее в главе.

Обычно компилятор уже на этапе компиляции решает, к какой именно функции обращается вызов. После того как вы щелкаете на кнопке, которая дает указание компилятору C++ (причем неважно — GNU C++ или, например, Visual C++) пересобрать программу, компилятор должен просмотреть ее и на основе используемых аргументов выбрать, какую именно перегружаемую функцию вы имели в виду.

В данном случае объявленный тип аргумента функции fn() не полностью описывает требования к функции. Хотя аргумент и объявлен как Student, он может оказаться также и GraduateStudent. Окончательное решение можно принять, только когда программа выполняется (это называется "на этапе выполнения"). И только когда функция fn() уже вызвана, C++ может посмотреть на тип аргумента и решить, какая именно функция-член должна вызываться: из класса Student или из GraduateStudent.



Типы аргументов, с которыми вы сталкивались до этого времени, называются объявленными или типами этапа компиляции. Объявленным типом аргумента x в любом случае является Student, поскольку так написано в объявлении функции fл(). Другой, текущий, тип называется типом этапа выполнения. В случае с примером функции fn() типом этапа выполнения аргумента x является Student, если fn() вызывается с s, и GraduateStudent, когда fn() вызывается с gs. Все понятно?

Способность решать на этапе выполнения, какую именно из нескольких перегружаемых функций в зависимости от текущего типа следует вызывать, называется *полиморфизмом* или поздним связыванием. Термин *полиморфизм* восходит к двум греческим корням — *поли* (т.е. многообразие) и *морф* (форма), дополненным широко известным греко-латинским суффиксом *изм.* C++ поддерживает полиморфизм. (Что и не удивительно. Тратил бы я столько времени на обсуждение полиморфизма, если бы он не поддерживался C++!) Чтобы подчеркнуть противоположность позднему связыванию, выбор перегружаемой функции на этапе компиляции называют ранним связыванием.



Полиморфизм и позднее связывание — не совсем эквивалентные термины. Полиморфизм означает способность выбора из возможных вариантов на этапе выполнения, тогда как позднее связывание — это всего лишь механизм, который используется языком C++ для осуществления полиморфизма. Разница здесь довольно тонкая.

Перегрузка функции базового класса называется переопределением (overriding). Такое новое название используется, чтобы отличать этот более сложный случай от нормальной перегрузки.

Зачем нужен полиморфизм

Полиморфизм является ключом (одним из связки), который способен открыть всю мощь объектно-ориентированного программирования. Он настолько важен, что языки, не поддерживающие полиморфизм, не имеют права называться объектно-ориентированными.



Языки, которые поддерживают классы, но не поддерживают полиморфизм, называются объектно-основанными. К таким языкам относится, например, Ada.

Без полиморфизма от наследования было бы мало толку. Позвольте привести еще один пример, чтобы вы поняли, почему это так. Представим себе, что я написал действительно сложную программу, использующую некий класс, который называется — не будем далеко ходить за примером — student. После нескольких месяцев разработки, кодирования и тестирования я выставляю эту программу на всеобщее обозрение, чтобы услышать восторженные отзывы и критику от свих коллег. (Программа настолько "крута", что уже заходит речь о передаче мне контрольного пакета акций Microsoft... но не будем опережать события.)

Проходит время, и мой босс просит добавить в программу возможность работы с аспирантами, которые хотя и очень похожи, но все-таки не идентичны обычным студентам (правда, аспиранты думают, что они совсем не похожи на студентов!). Мой босс не знает и не интересуется тем, что где-то глубоко в программе функция someFunction () вызывает функцию-член calcTuition () (такая уж работа у босса — ни о чем не думать и не волноваться...).

```
void someFunction(Student& s)
{
    //...то, что эта функция должна делать...
    s.calcTuition();
    //...функция продолжается..-
}
```

Если бы C++ не поддерживал позднее связывание, мне бы пришлось отредактировать функцию someFunction{) приблизительно так, как приведено ниже, и добавить ее в класс GradusteStudent.

```
#define STUDENT 1
#define GRADUATESTUDENT 2
void someFunction(Student& s)
i
    //...то, что эта функция должна делать...
    //добавим тип члена, который будет
    //индицировать текущий тип объекта
    switch (s.type)
    {
        case STUDENT:
            s.Student::calcTuition();
            break;
        case GRADUATESTUDENT:
            s.GraduateStudent::calcTuition();
            break;
    }
    //...функция продолжается...
}
```

Мне бы пришлось добавить в класс переменную type. После этого я был бы вынужден добавить присвоения type = STUDENT и type = GRADUATESTUDENT к конструктору GraduateStudent. Значение переменной type отражало бы текущий тип объекта s. Затем мне бы пришлось добавить проверяющие команды, показанные в приведенном выше фрагменте программы, везде, где вызываются переопределяемые функции.

Это не так уж и трудно, если не обращать внимания на три вещи. Во-первых, в данном примере описана только одна функция. Представьте себе, что calcTuition () вызывается из нескольких мест и что этой функции придется выбирать не между двумя, а между пятью или десятью классами. Маловероятно, что я найду все места в программе, которые надо отредактировать.

Во-вторых, я должен изменить (читай — сломать) код, который был отлажен и работал, а местами был довольно запутан. Редактирование может занять много времени и стать довольно скучной процедурой, что обычно ослабляет мое внимание. Любое изменение может оказаться ошибочным и конфликтовать с существующим кодом. Кто знает?..

И наконец, после того как я завершу редактирование, отладку и тестирование программы, я должен буду поддерживать две ее версии (если, конечно, не перестану поддерживать исходную). Это означает наличие двух потенциальных источников проблем в случае выявления ошибок и необходимость отдельной системы систематизации (как вам такая тавтология?), чтобы содержать все это в порядке.

А теперь представьте себе, что случится, когда мой босс захочет добавить еще один класс (босс — он такой: на все способен...). Мне придется не только повторить весь процесс сначала, а поддерживать три версии программы!

При наличии полиморфизма все, что потребуется сделать, — это добавить новый подкласс и перекомпилировать программу. В принципе мне может понадобиться изменить сам базовый класс, но только его и только в одном месте. Изменения в коде приложения будут сводиться к минимуму.

На некотором философском уровне есть еше более важные причины для полиморфизма. Помните, как я готовил закуски в микроволновой печи? Можно сказать, что я действовал по принципу позднего связывания. Рецепт был таким: разогрейте закуску в печи. В нем не было сказано: если печь микроволновая, сделай так, а если конвекционная — эдак. В рецепте (читай — коде) предполагалось, что я (читай — тот, кто осуществляет позднее связывание) сам решу, какой именно разогрев (функцию-член) выбрать, в зависимости от типа используемой печи (отдельного экземпляра класса Oven) или ее вариаций (подклассов), например таких, как микроволновая печь (Microvawe). Так думают люди, и так же создаются языки программирования: чтобы дать людям возможность, не изменяя образа мыслей, создавать более точные модели реального мира.

Как работает полиморфизм

С++ поддерживает и раннее и позднее связывание; однако вы, наверное, удивитесь, узнав, что в С++ по умолчанию используется раннее связывание. Если немного подумать, причина становится понятной. Полиморфизм требует несколько больше ресурсов (времени и памяти) для каждого вызова функции. Отцы-основатели С++ беспокоились о том, что любое изменение, которое они представят в С++ как усовершенствование его предшественника С, может стать поводом для неприятия этого языка в качестве системного языка программирования. Поэтому они сделали более эффективное раннее связывание используемым по умолчанию.

Чтобы сделать функцию-член полиморфной, программист на С++ должен пометить ее ключевым словом virtual так, как это показано ниже.

```
#include <iostream.h>
class Base
{
public:
    virtual void fn()
    {
        cout << "Мы в классе Base\n";
    }
};

class Subclass : public Base
{
public:
    virtual void fn()
    {
        cout << "Мы в классе SubClass\n";
    }
};

void test(Base& b)
{
    b.fn();    //Используется позднее связывание
}

int main(int argcs, char* pArgs[])
{
    Base bc;
    Subclass sc;
    cout << "Вызываем функцию test(bc)\n";
    test(bc);
    cout << "Вызываем функцию test(sc)\n";
    test(sc);
    return 0;
}
```

Ключевое слово virtual сообщает C++ о том, что fn() является полиморфной функцией-членом. Это так называемое виртуальное объявление fn() означает, что ее вызовы будут связаны позже, если есть хоть какие-то сомнения по поводу типа аргумента, с которым будет вызываться функция fn() на этапе выполнения.

В приведенном фрагменте fn() вызывается через промежуточную функцию test(). Когда функции test() передается объект базового класса, b.fn() вызывает функцию Base::fn(). Но когда функции test() передается объект класса Subclass, этот же вызов обращается к функции Subclass::fn().

Запуск программы приведет к выводу на экран таких строк:

```
Вызываем функцию test(bc)
Мы в классе Base
Вызываем функцию test(sc)
Мы в классе SubClass
```



Если вы уже освоились с отладчиком вашей среды C++, настоятельно рекомендую выполнить этот пример в пошаговом режиме.



Достаточно объявить функцию виртуальной только в базовом классе. Виртуальность наследуется подклассами автоматически. Однако в этой книге я следую стандарту кодирования, в соответствии с которым функции объявляются виртуальными везде.

Полиморфное приготовление закуски

Поскольку вы уже познакомились с кое-какими деталями объявления виртуальных функций, вернемся к примеру с закусками и посмотрим, как они выглядят в программе. Рассмотрим приведенный ниже фрагмент кода.

```
#include <dos.h>  //нужно для использования функции sleep()
class  Stuff!) {};
class Nachos  : public Stuff {};  //Это и есть наше блюдо

// Обычная печь
class Oven
{
public:
    virtual void cook(Nachos& nachos);
    //необходимые нам функции поддержки
    void turnOn();          //включить печь
    void turnOff();         //выключить печь
    void insert(Stuff& s) ; //поместить блюдо внутрь
    void remove(Stuff& s); //вынуть блюдо

protected:
    float temp;
};
void Oven::cook(Nachos& nachos)
{
    //разогреть печь  (включить и ждать с помощью цикла)
    //пока блюдо разогреется до температуры 350°)
    turnOn();
    while  (temp<350)  {}
    //поместить блюдо внутрь на 15 минут
    insert(nachos);
    sleep(15*60);

    //вынуть и выключить печь
    remcve(nachos);
    turrOff();
}
class Microwave  : public Oven
{
public:
    virtual void cook(Nachos& nachos);
    void rotateStuff(Stuff& s);
};
void Microwave::cook(Nachos& nachos)
{
    //никакого разогрева — температура не нужна
    //сначала поместить блюдо внутрь, затем включить
    insert(nachos);
    turnOn();

    //готовить только минуту
    // (через полминуты позернуть блюдо)
    sleep(30);            //подождать 30 секунд
    rotateStuff(nachos);
    sleep(30);            //подождать 30 секунд
```

```
    //сначала выключить печь, затем вынуть блюдо
    turnOff();
    remove(nachos);
}

Nachos makeNachos(Oven& oven)
f
    //Смешать ингредиенты
    Nachos n;

    //теперь приготовить блюдо
    //в любой печи, которая у вас есть
    oven.cook(n);

    //Съесть результат...
    return n;
}
```

Здесь вы видите класс Nachos, который объявлен как подкласс stuff (Блюдо). Класс Oven укомплектован общими функциями — turnOn (), turnoff (), insert () и remove () (с помощью последних двух блюдо помешается в печь и вынимается из нее). Кроме того, класс Oven содержит функцию-член cook (Nachos&), которая объявлена виртуальной.

Функция cook (Nachos&) была объявлена виртуальной, поскольку в подклассе Microwave, который наследуется от класса Oven, она работает по-другому. Функция Oven: :cook(Nachos&) разогревает печь до температуры 350°, помешает закуски внутрь и готовит их 15 минут. Затем оставшиеся от блюда угольки извлекаются из печи. В отличие от представления класса Oven о приготовлении закусок, функция Microwave: : cook (Nachos&) помещает блюдо внутрь, включает печь на 30 секунд, поворачивает блюдо и ждет еше 30 секунд, прежде чем выключить печь и вынуть блюдо.

Все это хотя и превосходно, но всего лишь присказка, сказка — впереди. Функции makeNachos () передается объект Oven некоторого типа. Получив такую oven (печь), она собирает все ингредиенты в объект п и готовит его, вызывая oven.cook (). Какая именно функция используется для этого — Oven : : cook () или Microwave : : cook () , зависит от текущего типа печи. Функция makeNachos (} не имеет об этом представления, да ее это и не интересует.

Так чем же так хорош полиморфизм? Во-первых, он возлагает заботу о подробностях работы печи на ее изготовителя, а не на того, кто в ней готовит. Такое разделение труда позволяет переложить детали работы печи на плечи программиста, специализирующегося по печам.

Во-вторых, полиморфизм может значительно упростить код. Посмотрите на то, как просто, без использования каких-либо подробностей работы микроволновой печи реализована функция makeNachos!) (я понимаю, что в данном случае эта функция не была бы намного сложнее с деталями, но ведь полиморфизм работает не только с печами!). И наконец, когда появится новый подкласс печей ConvectionOven со своей функцией-членом ConvectionOven::cook(Nachos&), не нужно будет ничего менять в makeNachos {), чтобы воспользоваться новой печью. Полиморфизм автоматически вызовет новую функцию при необходимости.

Что и говорить, полиморфизм, раскрывающий перед нами всю мощь наследования, — отличная вещь!

Когда функция не является tfufuni/альной

Даже если вы считаете, что каждая функция вызывается с использованием позднего связывания, это отнюдь не означает, что так и есть на самом деле.



C++ на этапе компилирования никак не указывает, какое связывание было использовано — ранее или позднее.

Для осуществления вызова с поздним связыванием нужно следить за тем, чтобы все необходимые функции-члены были объявлены идентично, включая возвращаемый тип. Если функция-член объявлена в подклассе с другими аргументами, она не будет переопределена как полиморфная, независимо от того, объявлена она виртуальной или нет. Например, измените приведенную выше функцию так, чтобы ее аргументы не совпадали с аргументами функции базового класса, и вновь выполните программу.

```
#include <iostream.h>
class Base
{
public:
    virtual void fn(int x)
    {
        cout << "Мы в классе Base, int x = " << x << "\n";
    }
};
class SubClass : public Base
{
public:
    virtual void fn(float x)
    {
        cout « "Мы в классе float x = " « x « "\n";
    }
};

void test(Base& b)
{
    int i = 1;
    b.fn(i);    //Здесь не используется позднее связывание
    float f = 2.0F;
    b.fn(f);     //И здесь тоже не используется
}

int main(int argcs, char* pArgs[])
{
    Ease bc;
    Subclass sc;
    cout << "Вызываем функцию test(bc)\n";
    test;(bc);
    cout << "Вызываем функцию test(sc)\n";
    test(sc);
    return 0;
}
```

Единственное отличие между этой и предыдущей программой в том, что функция fn () в классе Base объявлена как fn(int), тогда как в версии класса Subclass она объявлена как fn(float). Никакой ошибки это не вызовет, поскольку программа полностью корректна. Однако результаты не показывают никаких признаков полиморфизма:

```
Вызываем функцию test(bc)
Мы в классе Base, int x = 2
Мы в классе Base, int x = 2
Вызываем функцию test(sc)
Мы в классе Base, int x = 1
Мы в классе 3ase, int x = 2
```

Поскольку первый вызов передает значение типа int, не удивительно, что компилятор вызывает fn(int) как с bc, так и *с* sc. Несколько неожиданно то, что во втором вызове float конвертируется в int и при втором обращении к функции test () вызывается та же функция Base: : fn(). Это происходит потому, что объект ь, который передается функции test (), является объектом класса Base. Без полиморфизма вызов b.fn() Btest{) обращается к Base: : fn (int).



Если аргументы не полностью совпадают, позднее связывание не используется.

В правиле об идентичности объявления есть только одно исключение, которое состоит в том, что если функция-член базового класса возвращает указатель или ссылку на объект базового класса, то переопределяемая функция-член может возвращать указатель или ссылку на объект подкласса. Другими словами, приведенная ниже программа допустима.

```
class Base
{
public:
    //возвращаем копию текущего объекта
    Base* makeACopy()
    {
        //...делает все, что нужно для создания копии
    }
};

class Subclass : public Base
1
public:
    //возвращаем копию текущего объекта
    SubClass* makeACopy()
    {
        //...делает все, что нужно для создания копии
    )
};
void fn(BaseClass& bc)
{
    BaseClass* pCopy = bc.makeACopy();
    //функция продолжается. . .
}
```

С практической точки зрения все естественно: функция копирования гаакеСоруО должна возвращать указатель на объект типа Subclass, даже если она переопределяет Base::makeCopy().

Виртуальные особенности

При использовании виртуальных функций следует не забывать о некоторых вещах.

Во-первых, статические функции-члены не могут быть объявлены виртуальными. Поскольку статические функции-члены не вызываются с объектом, никакого объекта этапа выполнения не может быть, а значит, нет и его типа.

Во-вторых, при указании имени класса в вызове функция будет компилироваться с использованием раннего связывания независимо от того, объявлена она виртуальной или нет.

Например, приведенный ниже вызов обращается к Base::fn(], поскольку так указал программист, независимо от того, объявлена fn () виртуальной или нет.

```
void test (Bases b)
{
    b.base::fn();
    //Этот вызов не использует позднего связывания
}
```

Кроме того, виртуальная функция не может быть встроенной. Чтобы подставить функцию на место ее вызова, компилятор должен знать ее на этапе компиляции. Таким образом, независимо от способа описания виртуальные функции-члены рассматриваются как не встроенные.

И наконец, конструкторы не могут быть виртуальными, поскольку во время работы конструктора не существует завершенного объекта какого-либо определенного типа. В момент вызова конструктора память, выделенная для объекта, является просто аморфной массой. И только после окончания работы конструктора объект становится экземпляром класса в полном смысле этого слова.

В отличие от конструктора, деструктор может быть объявлен виртуальным. Более того, если он не объявлен виртуальным, вы рискуете столкнуться с неправильной ликвидацией объекта, как, например, в следующей ситуации:

```
class Base
f
    public:
    -Base();
};
class Subclass : public Base
{
public:
};  ~SubClass();

void finishWithObject(Base* pHeapObject)
{
    delete pHeapObject;  //Здесь вызывается ~Base()
                         //независимо от типа
                         //указателя pHeapObject
)
```

Если указатель, передаваемый функции finishWithObject (), на самом деле указывает на объект SubClass, деструктор Subclass все равно вызван не будет: поскольку он не был объявлен виртуальным, используется раннее связывание. Однако, если объявить деструктор виртуальным, проблема будет решена.

А если вы не хотите объявлять деструктор виртуальным? Тому может быть только одна причина: виртуальные функции несколько увеличивают размер объекта. Когда программист определяет первую виртуальную функцию в классе, C++ прибавляет

к классу дополнительный скрытый указатель — именно один указатель на класс! Класс, который не содержит виртуальных функций и не наследует никаких виртуальных функций от базовых классов, не будет содержать этого указателя. Однако один указатель не такая уж большая цена безопасной работы программы.



Лучше всегда объявлять деструкторы виртуальными, даже если ваш класс не наследуется (пока не наследуется!): ведь никогда не известно, в какой момент появится некто (может, это будете вы сами), желающий воспользоваться вашим классом как базовым для своего собственного класса. Если вы не объявили деструктор виртуальным, обязательно документируйте это!

Глава 23

Разложение классов

В этой главе...

- ✓ Разложение
- ✓ Реализация абстрактных классов
- ✓ Рационализация бюджета: BUDGET3.CPP

Концепция наследования позволяет классу наследовать свойства базового класса. Наследование помогает в достижении многих целей; например, благодаря ему я плачу за обучение моего сына. Оно помогает избежать повторения кода и сократить время, затрачиваемое на написание программ. Благодаря наследованию можно повторно использовать уже существующий код в новых программах, переопределяя функции.

Главное преимущество наследования — возможность указывать тип взаимосвязи между классами. Это так называемая взаимосвязь типа ЯВЛЯЕТСЯ: микроволновая печь ЯВЛЯЕТСЯ печью и т. д.

Разложение — это прекрасный способ создания правильных связей. К примеру, связь микроволновой печи с конвекционной печью кажется естественной. Утверждение же о том, что микроволновая печь является особым типом тостера, скорее всего, вас несколько насторожит. Конечно, оба эти прибора нагревают, оба используют электричество и оба находятся на кухне, но на этом сходство заканчивается — микроволновая печь не готовит тосты.

Процедура определения классов, свойственных данной проблеме, и задания корректных связей между этими классами известна под названием *разложение* (factoring) (это слово относится к арифметике, с которой вы мучились в средней школе; помните, как вы занимались разложением числа на простые множители: 12 равно 2, умноженное на 2 и на 3...).

Разложение

Чтобы увидеть, как работает разложение, вернемся назад и посмотрим на классы Checking и Savings, использованные в программе BUDGET, которая приводится в конце каждой части. Я мог бы до посинения рассказывать об этих классах, однако, к счастью, объектно-ориентированные программисты придумали довольно наглядный и краткий путь описания классов. Классы Checking и Savings показаны на рис. 23.1.

Для того чтобы правильно понять этот рисунок, необходимо знать несколько правил.

- ✓ Большой прямоугольник — это класс. Имя класса написано сверху.
- ✓ Имена в меньших прямоугольниках — это функции-члены.
- ✓ Имена не в прямоугольниках — это данные-члены.
- ✓ Имена, которые выступают за пределы прямоугольника, ограничивающего класс, являются открытыми; к этим членам могут обращаться функции, не являющиеся членами класса или его наследников. Члены, которые находятся полностью внутри прямоугольника, недоступны снаружи класса.
- ✓ Толстая стрелка обозначает связь типа ЯВЛЯЕТСЯ.
- ✓ Тонкая стрелка обозначает связь типа СОДЕРЖИТ.

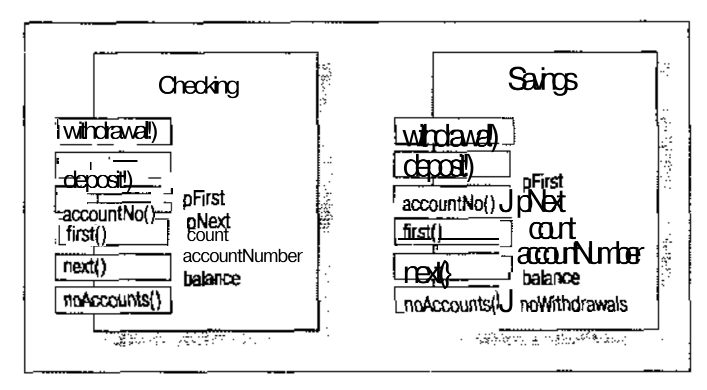


Рис. 23.1. Независимые классы Checking и Savings



Автомобиль ЯВЛЯЕТСЯ транспортным средством и при этом СОДЕРЖИТ мотор.

На рис. 23.1 вы можете увидеть, что классы Checking и Savings имеют много общего. Например, оба класса включают функции-члены withdrawal() и deposit(). Поскольку эти классы не идентичны, они, конечно же, должны оставаться раздельными (в реальном банковском приложении эти два класса отличались бы гораздо существеннее). Однако мы должны найти способ избежать дублирования.

Можно сделать так, чтобы один из этих классов наследовал другой. Класс Savings имеет больше членов, чем Checking, так что мы могли бы унаследовать Savings от Checking. Такой путь реализации этих классов приведен на рис. 23.2. Класс Savings наследует все члены класса Checking. Кроме того, в классе добавлен член noWithdrawal и переопределена функция withdrawal(). Эта функция переопределена, поскольку правила снятия денег со сберегательного счета отличаются от правил снятия с чекового счета (хотя меня эти правила вообще не касаются, поскольку у меня нет денег, которые можно было бы снять со счета).

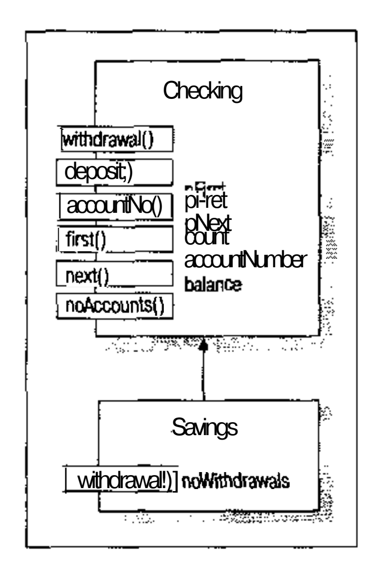


Рис. 23.2. Класс Savings реализован как подкласс Checking

Хотя наследование Savings от Checking и сберегает наш труд, нас оно не очень удовлетворяет. Главная проблема состоит в том, что оно искажает истинное положение вещей. При таком использовании наследования подразумевается, что счет Savings является специальным случаем счета Checking.

"Ну и что? — скажете вы. — Такое наследование работает и сохраняет нам силы и время". Это, конечно, так, но мои предупреждения — это не просто сотрясание воздуха. Такие искажения запутывают программиста уже и сейчас, но еще больше будут мешать в дальнейшем. Однажды программист, не знакомый с нашими "приемчиками", будет читать нашу программу, пытаясь понять, что же она

делает. Вводящие в заблуждение представления очень трудны для понимания и ведения программы.

Кроме того, такие искажения могут привести к проблемам в будущем. Например, представьте себе, что банк изменит свою политику относительно чековых счетов. Скажем, он решит взимать гонорар за обслуживание чековых счетов только в том случае, если минимальный баланс упадет ниже некоторого значения в течение месяца.

Такое изменение политики банка можно легко отразить в классе checking. Все, что нужно сделать, — это добавить новый член в класс checking, чтобы следить за минимальным балансом в течение месяца. Назовем его minimumBalance.

Однако теперь возникает проблема. Если Savings наследует Checking, значит, Savings тоже получает этот член. При этом он не используется, поскольку в сберегательных счетах минимальный баланс не нужен. Так что дополнительный член просто присутствует в классе. Итак, каждый объект чекового счета имеет дополнительный член minimumBalance. Один дополнительный член — это не так уж и много, но он вносит свою лепту в общую неразбериху.

Такие изменения имеют свойство накапливаться. Сегодня это один член, а завтра — измененная функция-член. В результате объекты класса Savings будут содержать множество дополнительных данных, которые нужны исключительно в классе Checking. Если вы будете невнимательны, изменения в классе Checking могут перейти к классу Savings и привести к его некорректной работе.

Как же этого избежать? Если поменять местами Checking и Savings, проблема не исчезнет. Нужен некий третий класс (назовем его Account), который будет воплощать в себе все то общее, что есть у Checking и Savings. Такая связь приведена на рис. 23.3.

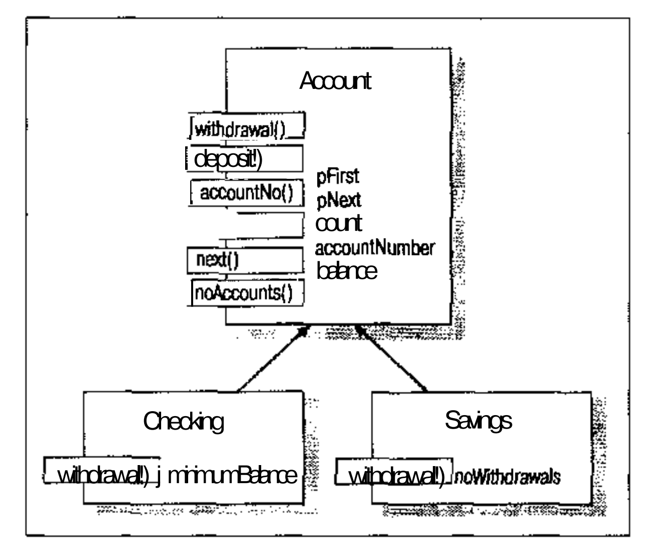


Рис. 23.3. Классы Checking и Savings, базирующиеся на классе Account

Каким образом создание нового класса Account решит наши проблемы? Во-первых, такой класс сделает более аккуратным описание реального мира (чем бы он ни являлся). В нашей концепции мира (по крайней мере, в моей) действительно есть что-то, могущее назваться счетом. Сберегательные и чековые счета являются частным случаем этой более фундаментальной концепции.

Кроме того, класс Savings отмежевывается от изменений в классе Checking (и наоборот). Если банк решит провести фундаментальные изменения во всех счетах, можно просто изменить класс Account, и все подклассы автоматически

унаследуют эти изменения. Но если банк изменит политику только для чековых счетов, можно просто модифицировать класс Checking, не изменяя при этом класс Savings.

Такая процедура отбора общих свойств похожих классов и называется *разложением*. Этот процесс очень важен в объектно-ориентированных языках по причинам, которые были приведены выше, а также потому, что разложение помогает избавиться от избыточности. Позвольте мне повториться: избыточность — это не просто плохо, это очень плохо...



Разложение будет обоснованным только в том случае, когда взаимосвязь, представляемая наследованием, соответствует реальности. Выделение общих свойств класса Mouse и Joystick и разложение их на "множители" вполне допустимо. И мышь и джойстик являются аппаратными устройствами позиционирования. Но выделение общих свойств классов Mouse и Display ничем не обосновано.

Разложение может давать (и обычно дает) результат на нескольких уровнях абстракции. Например, программа, написанная для более "продвинутого" банка, может иметь структуру классов, показанную на рис. 23.4.

Из этого рисунка видно, что между классами Checking и Savings и более общим классом Account вставлен еще один класс. Он называется Conventional и объединяет в себе особенности обычных счетов. Другие типы счетов, например счета ценных бумаг и биржевые счета, также объявляются как отдельные классы.

Такая многослойная структура классов весьма распространена и даже желательна (пока отношения, которые она представляет, отражают реальность. Однако не забывайте, что для любого заданного набора классов не существует одной единственно правильной иерархии классов).

Представим, что банк позволяет держателям счетов удаленно обращаться к чековым счетам и счетам ценных бумаг. Снимать же деньги с других типов счетов можно только в банке. Хотя структура классов, приведенная на рис. 23.4, выглядит естественной, в данных условиях более приемлема другая структура (рис. 23.5). Программист должен решить, какая структура классов лучше всего подходит к данным условиям, и стремиться к наиболее ясному и естественному представлению.

Реализация абстрактных классов

Такое интеллектуальное упражнение, как разложение, поднимает еще одну проблему. Вернемся к классам банковских счетов еще раз, а именно к общему базовому классу Account. На минуту задумайтесь над тем, как вы будете определять различные функции класса Account.

Большинство функций-членов класса Account не составят проблем, поскольку оба типа счета реализуют их одинаково. Однако функция Account.withdrawal() отличается в зависимости от типа счета. Правила снятия со сберегательного и чекового счетов различны. Мы вынуждены реализовывать Savings::withdrawal() не так, как Checking::withdrawal(). Но как реализовать функцию Account::withdrawal()?

Попросим банковского служащего помочь нам. Я так и представляю себе эту беседу:

"Каковы правила снятия денег со счета?" — спросите вы с надеждой.

"Какого именно счета, сберегательного или чекового?" — ответит он вопросом на вопрос.

"Со счета, — скажете вы, — просто со счета!"

Пустой взгляд в ответ...

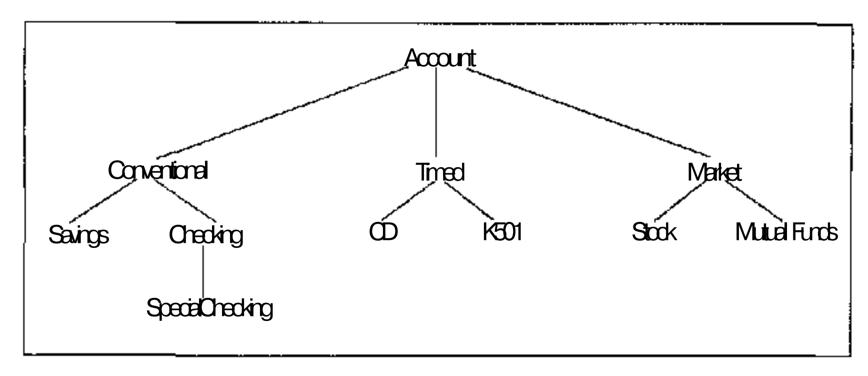


Рис. 23.4. Развитая структура банковских счетов

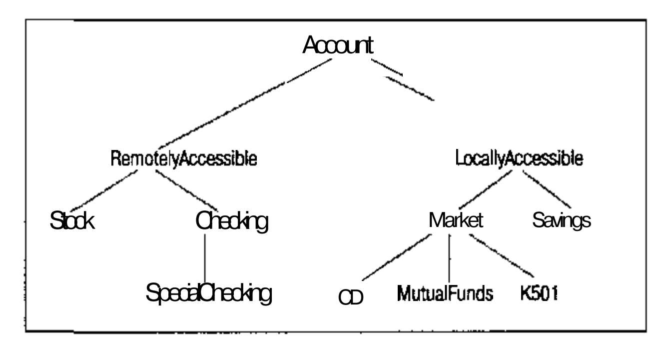


Рис. 23.5. Альтернативная иерархия классов

Проблема в том, что такой вопрос не имеет смысла. Нет такой веши, как "просто счет". Все счета (в данном примере) должны быть чековыми или сберегательными. Концепция счета — это абстракция, с помощью которой мы объединяем общие свойства для конкретных счетов. Это незавершенная концепция, поскольку в ней отсутствует такое важное свойство, как функции `withdrawal()` (если вы углубитесь в детали, то найдете и другие свойства, которых не хватает "просто счету").

Абстрактный класс — это тот класс, который реализуется только в подклассе. Конкретный — тот, который не является абстрактным.

Чтобы объяснить, что я имею в виду, позвольте позаимствовать пример из мира животных. Наблюдая разные особи теплокровных и живородящих, вы можете заключить, что они все укладываются в концепцию под названием "млекопитающие". Вы можете выделить такие классы млекопитающих, как собачьи, кошачьи и гуманоиды. Однако невозможно найти где-либо на земле просто млекопитающее. Другими словами, млекопитающие не могут содержать особь под названием "млекопитающее". Млекопитающее — это концепция высокого уровня, которую создал человек, и экземпляров-млекопитающих не существует.

Обратите внимание, что утверждать это с уверенностью я могу только по истечении некоторого времени. Ученые постоянно открывают новые виды животных. Проблема в том, что каждое существо обладает свойствами, которых не имеют другие; однако вполне вероятно, что в будущем кто-то найдет такое свойство у других существ.

Отражая эту ситуацию, C++ предоставляет возможность оставлять абстрактные классы незавершенными.

Концепция абстрактных классов

Абстрактный класс — это класс с одной или более чисто виртуальной функцией. Прекрасно, это все разъясняет...

Ну хорошо, *чисто виртуальная функция* — это функция-член без тела функции (которого нет. например, потому, что никто не знает, как реализовать это самое тело).

Бессмысленно спрашивать о том, каким должно быть тело функции withdrawal() в классе Account. Хотя, конечно, сама концепция снятия денег со счета имеет смысл. Программист на C++ может написать функцию withdrawal(), которая будет отражать концепцию снятия денег со счета, но при этом данная функция не будет иметь тела, поскольку мы не знаем, как ее реализовать. Такая функция называется чисто виртуальной[18] (не спрашивайте меня, откуда взялось это название).

Синтаксис объявления чисто виртуальной функции показан в приведенном ниже классе Account.

```
//Account — это абстрактный класс

class Account
{
protected:
    Account(Account& c);
public:
    Account(unsigned accNo, float initialBalance = 0.0F);

    unsigned int accountNo();
    float acntBalance();
    static Account *first();
    Account *next();
    static int noAccounts();

    //функции транзакций
    void deposit();

    //приведенная ниже функция является чисто виртуальной
    virtual void withdrawal(float amount) = 0;

protected:
    //если хранить счета в связанном списке, не
    //будет ограничения на их количество
    static Account *pFirst;
    Account *pNext;
    static int count;    //количество счетов
    unsigned accountNumber;
    float balance;
};
```

Наличие после объявления функции withdrawal() символов = 0 показывает, что программист не намеревается в данный момент определять эту функцию. Такое объявление просто занимает место для тела функции, которое позже будет реализовано в подклассах. От подклассов класса Account ожидается, что они переопределят эту функцию более конкретно.

---

[18] *Вообще говоря, чисто виртуальная функция может иметь тело, но обсуждение этого вопроса выходит за рамки данной книги.* — Прим. ред.



Я считаю это объяснение глупым, и мне оно нравится не более чем вам, так что просто выучите и живите с ним. Для этого объяснения есть причина, если не оправдание. Каждая виртуальная функция должна иметь свою ячейку в специальной таблице, в которой содержится адрес функции. Так вот: ячейка для чисто виртуальной функции содержит ноль.

Абстрактный класс не может быть реализован; другими словами, вы не можете создать объект абстрактного класса. Например, приведенное ниже объявление некорректно.

```
void fn()
{
    Account acnt(1234,100.00);  //это некорректно
    acnt.withdrawal(50);        //куда, по-вашему, должен
                                //обращаться этот вызов?
}
```

Если бы такое объявление было разрешено, конечный объект оказался бы незавершенным, поскольку был бы лишен некоторых возможностей. Например, что бы выполнял приведенный в этом же объявлении вызов? Помните, функции Account::withdrawal() не существует.

Абстрактные классы служат базой для других классов. Account содержит универсальные свойства для всех банковских счетов. Вы можете создать другие типы банковских счетов, наследуя класс Account, но сам этот класс не может быть реализован.

Создание полноценного класса из абстрактного

Подкласс абстрактного класса остается абстрактным, пока в нем не переопределены все чисто виртуальные функции. Класс Savings не является абстрактным, поскольку переопределяет чисто виртуальную функцию withdrawal() совершенно реальной. Объект класса Savings отлично знает, как реализовать функцию withdrawal() и куда обращаться при ее вызове. То же касается и класса Checking: он не виртуальный, поскольку withdrawal() переопределяет чисто виртуальную функцию, определенную ранее в базовом классе.

Подкласс абстрактного класса, конечно, может оставаться абстрактным. Разберемся с приведенными ниже классами.

```
class Display
{
public:
    virtual void initialize () = 0;
    virtual void write(char *pString) = 0;
};

class SVGA : public Display
{
    //сделаем обе функции-члена "реальными"
    virtual void initialize();
    virtual void write(char *pString);
};

class HWVGA : public Display
{
    //переопределим только одну функцию
```

```
    virtual void write(char *pString);
};

class ThreedVGA : public HWVGA
{
    virtual void initialized;
};

void fn()
{
    SVGA mc;
    VGA vga;
    //все остальное
}
```

Класс Display, описывающий дисплеи персонального компьютера, содержит две чисто виртуальные функции: initialize () и writef]. Вы не можете ввести эти функции в общем виде. Разные типы видеоадаптеров инициализируются и осуществляют вывод по-разному.

Один из подклассов — SVGA — не абстрактный. Это отдельный тип видеоадаптера, и программист точно знает, как его реализовать. Таким образом, класс SVGA переопределяет обе функции— initialized и write() — именно так, как необходимо для данного адаптера.

Еще один подкласс — HWVGA . Программисту известно, как программировать ускоренный VGA-адаптер. Поэтому между общим классом Display и его частным случаем, ThreedVGA, который представляет собой специальный тип карт 3-D, находится еще один уровень абстракции.

В нашем обсуждении предположим, что запись во все аппаратно ускоренные карты VGA происходит одинаково (это не соответствует истине, но представим себе, что это так). Чтобы правильно выразить общее свойство записи, вводится класс HWVGA, реализующий функцию write() (и другие общие для HWVGA свойства). При этом функция initialize () не переопределяется, поскольку для разных типов карт HWVGA она реализуется по-разному.

Поэтому, несмотря на то что функция write () переопределена в классе HWVGA, ОН все равно остается абстрактным, поскольку функция initialize () все еще не переопределена.

Поскольку ThreedVGA наследуется от HWVGA, он должен переопределить только одну функцию, initialize {), для того чтобы окончательно определить адаптер дисплея. Таким образом, функция fn () может свободно реализовать и использовать объект класса ThreedVGA.



Замещение нормальной функцией последней чисто виртуальной функции делает класс завершенным (т.е. неабстрактным). Только неабстрактные классы могут быть реализованы в виде объектов.



Изначально требовалось, чтобы каждая чисто виртуальная функция была переопределена в каждом подклассе другой, хотя бы и снова чисто виртуальной функцией. В конечном счете люди решили, что это глупое требование, и исключили его. Однако старые компиляторы могут требовать выполнения этого условия.

Передача абстрактных классов

Поскольку вы не можете реализовать абстрактный класс, упоминание о возможности создавать указатели на абстрактные классы звучит несколько странно. Однако если вспомнить о полиморфизме, то станет ясно, что это не так уж глупо, как кажется поначалу. Рассмотрим следующий фрагмент кода:

```
void fn(Account *pAccount);   //это допустимо
void otherFn ()
{
    Savings s;
    Checking c;

    //Savings ЯВЛЯЕТСЯ Account
    fn(&s) ;
    //Checking — тоже
    fn(sc);
}
```

В этом примере pAccount объявлен как указатель на Account. Разумеется, при вызове функции ей будет передаваться адрес какого-то объекта неабстрактного класса, например Checking или Savings.

Все объекты, полученные функцией fn (), будут объектами либо класса Checking, либо Savings (или другого неабстрактного подкласса Account). Можно с уверенностью заявить, что вы никогда не передадите этой функции объект класса Account, поскольку никогда не сможете создать объект этого класса.

Нужны ли чисто виртуальные функции

Если нельзя определить функцию withdrawal;), почему бы просто не опустить ее? Почему бы не объявить ее в классах Savings и Checking, где она может быть определена, оставив в покое класс Account? Во многих объектно-ориентированных языках вы могли бы именно так и сделать. Но C++ предпочитает иметь возможность убедиться в вашем понимании того, что вы делаете.



Не забывайте, что объявление функции — это указание полного имени функции, включающего ее аргументы. Определение же функции включает в себя и код, который будет выполняться в результате вызова этой функции.

Чтобы продемонстрировать суть сказанного, можно внести следующие незначительные изменения В класс Account:

```
class Account
{
    //то же, что и раньше, но
    //нет функции withdrawal()
};

class Savings : public Account
{
    public:
    virtual void withdrawal(float amnt);
};

void fn(Account *pAcc)
{
    //снять некоторую сумму
```

```
    pAcc->withdrawal(100.00f);
                        //этот вызов недопустим,
                        //поскольку withdrawal()
                        //не является членом класса Account
}

int main()
!
    Savings s;       //открыть счет
    fn(&s);
    //продолжение программы
}
```

Представьте себе, что вы открываете сберегательный счет s. Затем вы передаете адрес этого счета функции fn (}, которая пытается выполнить функцию withdrawal (). Однако, поскольку функция withdrawal () не член класса Account, компилятор сгенерирует сообщение об ошибке.

Некоторые языки выполняют такую проверку, когда функция уже вызвана, во время выполнения программы. В таком случае приведенный выше фрагмент кода будет работать: raain() будет вызывать fn () и передавать ей объект s. Когда fn(), в свою очередь, вызовет функцию withdrawal (), программа увидит, что withdrawal!) действительно определена в переданном ей объекте. Цена такой гибкости — снижение скорости выполнения программы, поскольку язык должен проводить множество проверок во время ее выполнения. Это также чревато некоторыми ошибками. Например, если кто-то передаст объект, который является счетом, но не содержит определенной в нем функции withdrawal О, то программа аварийно прекратит работу, так как не сможет определить, что делать с этим вызовом. Я думаю, пользователи будут не очень рады этому.

Взгляните, как чисто виртуальная функция помогает решить эту проблему. Вот та же ситуация с абстрактным классом Account:

```
class Account
{
    //почти то же, что и в предыдущей программе,
    //однако функция withdrawal() определена
    virtual void withdrawal(float amnt) = 0;
};

class Savings : public Account
{
public:
    virtual void withdrawal(float amnt);
};

void fn(Account *pAcc)
i
    //снять некоторую сумму
    //теперь этот код будет работать
    pAcc->withdrawal(100.00f);
}

int main()
{
    Savings s;    //открыть счет
    fn(&s);
    //продолжение программы
}
```

Ситуация та же, но теперь класс Account содержит функцию-член withdrawal {}. Поэтому, когда компилятор проверяет, определена ли функция pAcc->withdrawal(), он видит ожидаемое определение Account::withdrawal(). Компилятор счастлив. Вы счастливы. А значит, и я тоже счастлив. (Честно говоря, для того чтобы сделать меня счастливым, достаточно футбола и холодного пива.)

Чисто виртуальная функция занимает место в базовом классе для функции с тем, чтобы позже быть переопределенной в подклассе, который будет знать, как ее реализовать. Если место не будет занято в базовом классе, не будет и переопределения.

Рационализация бюджета: BUDGET3.CPP

В этой части продолжается превращение исключительно функциональной версии программы Budget1, приведенной в конце части 2, которая затем прошла через объектно-основанный этап своей эволюции — Budget2, представленную в конце части 3. Теперь она наконец превратится в объектно-ориентированную программу Budget3.

Программа Budget осуществляет вложение денег на счет и снятие со счета в воображаемом банке. Пользователь поочередно вводит номера банковских счетов и суммы вкладов на этот счет и снятий с него. После того как пользователь выполнил все транзакции, программа показывает баланс каждого счета и общий баланс. Обе программы — Budget2 и Budget3 — эмулируют Checking (чековый) и Savings (сберегательный) счета. Чековые счета взимают небольшой гонорар за обслуживание при каждом снятии, если баланс упал ниже 500 долларов, тогда как сберегательный счет взимает большой гонорар за обслуживание при первом снятии, независимо от баланса.

Программа Budget2 превосходит 3udgetl только в одном: она изолирует особенности классов, описывающих счет, от внешних функций, которые манипулировали счетами. К сожалению, Budget2 содержала большое количество дублированного кода в классах Savings и Checking, и именно от него мы и хотим избавиться, используя принципы наследования.

Программа Budget3 обладает следующими преимуществами:

- ✓ использует наследование, чтобы выделить общие для чековых и сберегательных счетов свойства, избегая избыточности;
- ✓ использует виртуальные функции-члены для улучшения читаемости и гибкости программы;
- ✓ создает чисто виртуальные классы для выделения общих черт, присущих чековым и сберегательным счетам;
- ✓ вместо массива использует связанный список во избежание ограничения на количество счетов, которые может поддерживать банк.

С помощью такого "супергероя" объектно-ориентированного программирования, как наследование, и его верного бокового удара — полиморфизма мы смогли оптимизировать два класса счетов, объединив в один класс Account все то общее, что присуще этим двум классам. В результате получилась гораздо более короткая и простая программа.

```
// BUDGET3.CPP — Программа банковского бюджета
//     с наследованием и полиморфизмом. Обратите
//     внимание на то, насколько меньше стала эта
//     программа по сравнению с Budget2,
//     после того как из нее был удален
//     дублированный код. Теперь одна функция
//     может обрабатывать и чековые и сберегательные
//     счета (а также любые другие, которые вы можете
//     придумать в будущем).
//
//     Кроме того, вместо массива, который может иметь
//     только определенную длину, эта версия хранит
//     счета в связанном списке.
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

class LinkedListObject
{
public:
    LinkedListObject()
    {
        // добавим текущий объект к
        // связанному списку
        addToEnd();
    }

    // управление связанным списком
    static LinkedListObject* first()
    {
        return pFirst;
    }
    LinkedListObject* next()
    {
        return pNext;
    }
    void addToEnd();
protected:
    // будем содержать счета в связанном списке,
    // так что теперь не будет ограничения на
    // количество объектов
    static LinkedListObject* pFirst;
    LinkedListObject* pNext;
};

// инициализируем указатель на первый
// объект в связанном списке
LinkedListObject* LinkedListObject::pFirst = 0;

// добавим текущий объект к связанному списку объектов
void LinkedListObject::addToEnd()
{

    // добавим объект this к концу списка
    // и учтем его в счетчике count
```

```
    if (pFirst == 0)
    {
        // список пуст; сделаем этот объект первым
        pFirst = this;
    }
    else
    {
        // найдем последний элемент списка
        LinkedListObject* pA;
        for (pA = pFirst; pA->pNext; pA = pA->pNext) {}
        // свяжем текущий объект с концом списка
        pA->pNext = this;
    }
    pNext = 0;          // этот объект — последний
}


// Account — этот абстрактный класс объединяет в себе
//    общие свойства обоих типов счетов: как чековых,
//    так и сберегательных. Здесь отсутствует реализация
//    withdrawal() - функции снятия, которая различна
//    для разных типов счетов
class Account : public LinkedListObject
{
public:
    Account::Account(unsigned accNo,
                     double initialBalance = 0.0)
    {
        // инициализируем данные-члены объекта
        accountNumber = accNo;
        balance = initialBalance;

        // учтем этот объект
        count++;
    }

    // функции доступа
    int accountNo()
    {
        return accountNumber;
    }
    double acntBalance()
    {
        return balance;
    }
    static int noAccounts()
    {
        return count;
    }

    // функции связанного списка, обеспечивающие
    // поддержку работы с ним. Они
    // помогут сберечь много времени в будущем
    static Account* first()
    {
        return (Account*)LinkedListObject::first();
    }
```

```
    Account* next()
    {
        return (Account*)LinkedListObject::next();
    }

    // функции транзакций

    void deposit(double amount)
    {
        balance += amount;
    }

    virtual void withdrawal(double amount) = 0;

    // функция вывода объекта
    void display()
        cout << type()
             << " счет номер " << accountNumber
             << " = "   << balance
             << "\n";
    }
    virtual char* type() = 0;

protected:
    static int count;    // количество счетов
    unsigned accountNumber;
    double   balance;
};

// выделим место под статический член
int Account::count = 0;

// Checking — этот класс содержит уникальные для данного
//    класса свойства. Много ли от него осталось?
class Checking : public Account
{
public:
    Checking::Checking(unsigned sccNo,
                       double initialBalance = 0.0)  :
    Account(accNo, initialBalance)
    {
    }

    // перегрузим чисто виртуальные функции
    virtual void withdrawal(double amount);
    char* type()
    f
        return "Чековый";
    }
};

// withdrawal — перегрузим функцию-член
//    Account::withdrawal() так, чтобы ока
//    взимала 20 центов за каждое снятие
//    с чека с балансом меньше $500
void Checking::withdrawal(double amount)
{
    if (balance < amount )
```

```
    {
        cout << "Недостаточно денег на счете: "
                "баланс равен " << balance
             << ", попытка снять " << amount
             << "\n";
    }
    else
    {
        balance -= amount;
        // если баланс упал слишком низко,
        // взимать гонорар за обслуживание
        if (balance < 500.00)
        {
            balance -= 0.20;
        }
    }
}

// Savings — то же самое, что и Checking, за
// исключением еще одного уникального члена
class Savings : public Account
{
public:

    Savings::Savings(unsigned accNo,
                     double initialBalance = 0.0) :
    Account(accNo, initialBalance)
    {
        noWithdrawals = 0;
    }

    // функции транзакций
    virtual void withdrawal(double amount);
    char* type()
    {
        return "Сберегательный";
    }

protected:
    int noWithdrawals;
};

// withdrawal - перегрузим функцию-член
//    Account::withdrawal() так, чтобы она
//    взимала гонорар за обслуживание в
//    размере $5 при первом снятии за месяц
void Savings::withdrawal(double amount)
{
    if (balance < amount)
    {
        cout << "Недостаточно денег на счете: "
                "баланс равен " << balance
             << ", попытка снять " << amount
             << "\n";
    }
    else
    {
```

```
        if (++noWithdrawals > 1)
        {
            balance -= 5.00;
        }
        balance -= amount;
    }
}

// объявление прототипов
unsigned getAccntNo();
void     process(Account* pAccount);
void     getAccounts();
void     displayResults();


// main — собирает входные данные и выводит общие суммы
int main(int argcs, char* pArgs[])
{
    // прочитать счета, введенные пользователем
    getAccounts();

    // показать связанный список счетов
    displayResults();
    return 0;
}

// getAccounts — загрузить указанный массив счетов
void getAccounts()
{
    Account* pA;

    // повторять цикл, пока не будет введено "x" или "X"
    char   accountType;     // S или C
    while (1)
    {
        cout << "Нажмите S для сберегательных счетов,\n"
             << "C для чековых, X для выхода: ";
        cin >> accountType;
        switch (accountType)
        {
            case 'c':
            case 'C':
                pA = new Checking(getAccntNo());
                break;
            case 's':
            case 'S':
                pA = new Savings(getAccntNo());
                break;
            case 'x':
            case 'X':
                return;
            default:
                cout << "Неверный ввод.\n";
        }

        // теперь обработаем только что созданный объект
        process(pA);
```

```
    }
}

// displayResults — отобразить все счета, которые
//                  есть в связанном списке счетов
void displayResults()
{
    // показать сумму
    double total = 0.0;
    cout << "Суммы по всем счетам:\n";
    for (Account* pA = Account::first();
         pA; pA = pA->next())
    {
        pA->display();
        total += pA->acntBalance();
    }
    cout << "Всего = " << total << "\n";
}

// getAccntNo — возвращает введенный номер счета
unsigned getAccntNo()
{
    unsigned accntNo;
    cout << "Введите номер счета: ";
    cin  >> accntNo;
    return accntNo;
}

// process(Account) — ввод данных для счета
void process(Account* pAccount)
{
    cout << "Введите положительную сумму для вклада, \n"
         << "отрицательную для снятия, "
            "0 для прекращения работы\n";
    double transaction;
    do {
        cout << ":";
        cin  >> transaction;

        // вклад
        if (transaction > 0)
        {
            pAccount->deposit(transaction);
        }
        // снятие
        if (transaction < 0) {
            pAccount->withdrawal(-transaction);
        }
    } while (transaction != 0) ;
}
```

Я запустил эту программу с приведенными ниже данными, чтобы продемонстрировать, как она работает {или, как это чаще случается с моими программами, не работает). Жирным шрифтом выделен пользовательский ввод, а обычным представлены сообщения программы.

```
Нажмите S для сберегательных счетов,
C для чековых, X для выхода: S
```

```
Введите номер счета: 123
Введите положительную сумму для вклада,
отрицательную для снятия, 0 для прекращения работы
:200
:-20
:0
Нажмите S для сберегательных счетов,
С для чековых, X для выхода: s
Введите номер счета: 234
Введите положительную сумму для вклада,
отрицательную для снятия, 0 для прекращения работы
:200
:-10
:-10
:0
Нажмите S для сберегательных счетов,
С для чековых, X для выхода: с
Введите номер счета: 345
Введите положительную сумму для вклада,
отрицательную для снятия, 0 для прекращения работы
:200
:-20
:0
Нажмите S для сберегательных счетов,
С для чековых, X для выхода: С
Введите номер счета: 456
Введите положительную сумму для вклада,
отрицательную для снятия, 0 для прекращения работы
:600
:-20
:0
Нажмите S для сберегательных счетов,
С для чековых, X для выхода: х
Суммы по всем счетам:
Сберегательный счет номер 123 = 180
Сберегательный счет номер 234 = 175
Чековый счет номер 345 = 179.8
Чековый счет номер 456 = 580
Всего = 1114.8
```

Объектно-ориентированная программа Budget3 начинается с базового класса `LinkedListObject`. Этот класс содержит члены, необходимые для создания связанного списка. Здесь есть указатели на первый и следующий объекты в связанном списке, а также открытые функции-члены `first()` и `next()`. Любой класс, полученный из `LinkedListObject`, может быть использован для создания связанного списка, поскольку он наследует все необходимые функции-члены.

Следующий класс, содержащийся в программе Budget3, — это класс Account. Он обобщает в себе все, что можно сказать о счетах, а именно:

- ✓ они распознаются по номерам;
- ✓ каждый счет имеет баланс;
- ✓ пользователь может вкладывать или снимать деньги со счета.

Нам известно, как выполнять вложение денег на счет, поэтому функция `deposit()` определена прямо в классе Account. Однако мы не знаем, как в общем

виде выполнить снятие денег, поскольку для разных типов счетов эта операция различна. Поэтому функция Account::withdrawal() объявлена чисто виртуальной (с помощью "= o" в конце объявления).

Конструктор класса Account начинается с автоматического вызова конструктора LinkedListObject, который добавляет текущий счет к концу связанного списка объектов Account. Затем конструктор Account создает уникальную для каждого счета информацию, записывая номер счета и начальный баланс (который приравнивается нулю, если при создании счета не был задан другой баланс). Затем увеличивается на единицу значение статического члена count, с помощью которого отслеживается количество существующих в данный момент объектов Account.



Для одного класса существует только одна копия каждого статического объекта. К ней имеют доступ все объекты класса.

Функции accountNo() и accountBalance() служат для того, чтобы предоставлять возможность считывания номера счета и информации о балансе из внешнего мира, но не допускать непосредственного изменения этих значений.

Функции-члены Account::first(; и next() переопределяют свои более ранние версии из класса LinkedListObject. Без этих на первый взгляд бессмысленных функций пользователь должен был бы сам приводить тип объектов LinkedListObject, которые возвращают функции first() и next(), к типу Account.

Функции display() и type() придают всем счетам одинаковый формат отображения.

Подкласс checking класса Account достаточно прост. Конструктор класса Checking не делает ничего, кроме передачи аргументов конструктору класса Account. Единственная настоящая функция-член в этом классе — это withdrawal O , которая реализует правила работы с чековыми счетами.

Класс Savings идентичен в этом классу Checking: все, что он делает, — это реализует метод withdrawal ().



Любой подкласс класса Account, который не переопределяет функцию withdrawal (), будет виртуальным, и вы не сможете создать объект этого класса.

Функции, составляющие главную программу, теперь упрощены до предела. Функция getAccount () создает счет класса Checking или Savings (в зависимости от символа, введенного пользователем). Это единственное место в программе, где происходит непосредственное обращение к подклассам класса Account.

Функция displayResult () просматривает связанный список, требуя от каждого объекта отобразить свое содержимое, не интересуясь при этом деталями того, как именно сберегательные или чековые счета (или любые другие) будут это выполнять.

Функция process!) имеет еще более впечатляющий вид. Она выполняет вложения (которые обрабатывает функция Account: :deposit()) и снятия (которые обрабатывают функции Savings::withdrawal() и Checking::withdrawal(), в зависимости от типа объекта, на который указывает pAccount).

Обратите внимание на то, какой привлекательной теперь стала функция process (). Во-первых, полностью исчезла избыточность, которая появлялась благодаря существованию разных версий этой функции. Более важно то, что логика функции process!) значительно упростилась. Теперь программист может

сконцентрироваться на работе этой функции, не волнуясь о внутренних особенностях разных типов счетов.

Задача, которую решает Budget3, довольно проста. Но сравнение разных версий программы Budget позволит понять отличия между чисто функциональной программой (Budget1) и ее развитием — через объектно-основанную программу без наследования (Budget2) в полностью объектно-ориентированную программу (Budget3).

Часть V

Полезные особенности



В этой части...

В этой книге не ставится цель сделать из вас профессионала в области C++, *а всего лишь предполагается дать вам твердое понимание основ* C++ *и объектно-ориентированного программирования. В предыдущих частях книги вы приобрели самые необходимые знания по созданию качественной объектно-ориентированной программы. Конечно оке,* C++ *весьма обширный и богатый разнообразными возможностями язык, и осталось еще немала особенностей, которые требуют освещения. В этой части представлено краткое описание дополнительных и, по моему мнению, наиболее полезных возможностей языка, которые стоит использовать в первую очередь (хотя это и не обязательно).*

Глава 24

Перегрузка операторов

В этой главе...

- ✓ Перегрузка операторов: давайте жить в гармонии
- ✓ Операторная функция
- ✓ А подробнее?
- ✓ Операторы как функции-члены
- ✓ Еще одна перегрузка
- ✓ Перегрузка операторов с помощью неявного преобразования типов

Специальные короткие символы, которые вы используете в С++ (+, -, & и т.п.) называются простыми операторами. Эти операторы (вернее, их действия) уже определены для таких встроенных типов, как int, double и char (для некоторых типов отдельные операторы не определены). Эти операторы, конечно же, не определены для классов, которые создали вы сами (эти классы называются пользовательскими).

Вероятно, нам повезло: С++ позволяет определять, что именно будут означать операторы, если их применить к пользовательским классам. Эта особенность, называемая перегрузкой операторов, и является темой данной главы.

Я не зря сказал "вероятно". Обычно перегрузка операторов не обязательна и, более того, противопоказана новичкам в программировании на С++. Следует также отметить, что многие опытные программисты на С++ считают перегрузку операторов не такой уж хорошей идеей. Так что, если вы не чувствуете себя в силах освоить эту тему, можете пропустить данную главу и вернуться к ней позже, когда вам станет интересно и вы почувствуете себя готовым понять ее.

Итак, я вас предупредил; тем не менее вы все-таки хотите знать, как перегружать оператор присвоения, а также операторы "<<" и ">>"... К счастью, существуют шаблоны, которым можно следовать для переопределения этих трех операторов (что значительно облегчает дело). Поскольку я не хочу, чтобы вы запутались в этих трех операторах, я не пожалел для каждого из них отдельной главы.



Позвольте повториться (мне не нравится получать гневные письма с упоминанием всей моей родословной из-за того, что кто-то перегрузил какой-то оператор и теперь не знает, как ему заставить программу снова работать): *перегрузка оператора может привести к ошибкам, которые очень трудно выявить.* Еще раз проверьте, знаете ли вы, что делаете, прежде чем что-то переопределять.

Перегрузка операторов: давайте жить в гармонии

Язык С++ предполагает, что пользовательские типы имеют такие же права, как и встроенные, например int или char. Коль скоро операторы определены для встроенных типов, почему бы не позволить определять их для пользовательских типов?

Я осознаю, что это слабый аргумент, тем не менее перегрузка операторов может быть полезной. Допустим, у нас есть класс USDollar, который представляет вечнозеленые американские доллары. Некоторые операторы не имеют никакого смысла в применении к долларам. Например, какую смысловую нагрузку будет нести инверсия (оператор ~) в применении к долларам? Перевернуть бумажку на другую сторону? Обменять на гривну? С другой стороны, некоторые операторы явно применимы к этому классу. Например, имеет смысл складывать или вычитать объекты класса USDollar — результатом будет тот же USDollar. Кроме того, имеет смысл умножать или делить USDollar на два. Или три. Но очень трудно себе представить умножение USDollar на USDollar...

Перегрузка простых арифметических операторов для класса USDollar может значительно улучшить читаемость программы. Сравните два приведенных ниже фрагмента кода:

```
//expense — посчитать количество уплаченных денег
//          (учитывая проценты и требуемую сумму)
USDollar expense(USDollar principle, double rate)
{
    //подсчитаем проценты
    USDollar interest = principle.interest(rate);

    //добавим это к требуемой сумме
    //и вернем результат
    returnprinciple.add(interest),-
}
```

Если перегрузить операторы, эта же функция примет такой вид:

```
//expense — посчитать количество уплаченных денег
//          (учитывая проценты и требуемую сумму)
USDollar expense(USDollar principle, double rate)
{
    USDollar interest = principle*rate;
    return principle + interest;
```

Ну как, красиво?

Однако, прежде чем вы научитесь перегружать операторы, вы должны понять взаимосвязь между оператором и функцией.

Операторная функция

На секунду задумайтесь над тем, что такое оператор? Ведь это не более чем встроенная функция со своеобразным синтаксисом. Например, какая разница между a+Ь и +(a,b)? Или, например, add(a,b)? Никакой. Между прочим, в некоторых языках сложение именно так и происходит.

C++ дает каждому оператору специальное функциональное имя, которое состоит из ключевого слова operator, следующего за ним символа оператора и расположенных после него соответствующих типов аргументов. Например, оператор +, который выполняет сложение двух переменных типа int и лает на выходе int, по сути, вызывается как функция: int operator+(int, int).

Оператор, складывающий целочисленные значения, отличается от оператора, складывающего значения типа double (он будет выглядеть как double operator+(double, double)). Это не так трудно понять, если вспомнить, что внутренний формат переменной типа int отличается от формата переменной типа double.

Вы не можете изобретать новые операторы либо изменять приоритет или формат существующих операторов. Кроме того, нельзя переопределять операторы для

встроенных типов. Вы также вряд ли сможете внятно объяснить, что такое сложение двух целочисленных значений (если, конечно, не имеете соответствующей ученой степени).

Приведенные ниже примеры демонстрируют, как могут быть определены операторы сложения и инкремента для класса USDollar (я мог бы реализовать такие же функции и для канадских долларов, но тогда мне бы понадобился канадец для придания комментариям канадского акцента).

```
// USDollar — объект, который содержит целое число долларов
//            плюс целое число центов. Сто центов равны
//            одному доллару
class USDollar
{
    friend USDollar operator+(USDollar&, USDollar&);
    friend USDollar& operator++(USDollar&);

public:
    USDollar(unsigned int d, unsigned int c);

protected:
    unsigned int dollars;
    unsigned int cents;
};

// конструктор
USDollar::USDollar(unsigned int d, unsigned int c)
{
    dollars = d;
    cents = c;
    while (cents >= 100)
    {
        dollars++;
        cents -= 100;
    }
}

// operator+ — складывает s1 с s2 и возвращает
//             результат в виде нового объекта
USDollar operator+(USDollar& s1, USDollar& s2)
{
    unsigned int cents  = s1.cents  + s2.cents;
    unsigned int dollars = s1.dollars + s2.dollars;
    USDollar d(dollars, cents);
    return d;
}

// operator++ — увеличивает указанный аргумент,
//              изменяя значение объекта
USDollar& operator++(USDollar& s)
{
    s.cents++;
    if (s.cents >= 100)
    {
        s.cents -= 100;
        s.dollars++;
    }
    return s;
}
```

```
int main(int argcs, char* pArgs[])
{
    USDollar d1(1, 60);
    USDollar d2(2, 50);
    USDollar d3(0, 0);
    d3 = d1 + d2;
    ++d3;
    return 0;
}
```

Класс USDollar определен как класс, содержащий целое число долларов и целое число центов. Количество центов не должно превышать 100. Конструктор вводит еще одно правило, в соответствии с которым количество центов уменьшается на сто, если оно больше этой цифры, а количество долларов соответственно увеличивается.

В данном примере operator+ и operator++ были реализованы как обычные внешние функции, которые являются друзьями класса USDollar.

Скажи мне, кто твой друг...

Вы уже встречались со словом friend в главе 16, "Защищенные члены класса: не беспокоить!". Скажу о нем несколько слов, чтобы вам не пришлось лишний раз листать книгу. Внутри класса вы можете объявить внешнюю функцию как друга класса. Дружественная функция имеет все права и привилегии, которые присущим членам класса. Объявляя функцию operator+ () другом, я предоставляю ей доступ к защищенным членам класса USDollar.

Сравнение класса с семьей имеет тот же смысл, что и в нашей дискуссии об управлении доступом к классу. Например, все члены семьи имеют доступ к фамильному серебру (кроме сумасшедшей тетушки, однако не будем об этом). Большинству людей, за исключением тех, кого семья выбрала своими друзьями, не позволено прикасаться к фамильному серебру. В этом случае друг семьи имеет доступ к фамильному серебру. И первое, в чем семья доверяет другу, — это в том, что он не будет злоупотреблять доверием.

Заметьте, что человек не может объявить себя другом семьи; только семья может решить, друг он или нет. То же применимо и к классам: функция не может объявить себя другом класса. Таким образом, ключевое слово friend имеет смысл только внутри объявления класса.

Одна семья может предложить другой семье быть их друзьями. Это значит, что каждый член второй семьи имеет доступ к семейному серебру первой. Точно так же один класс может объявить другой класс другом, и это будет означать, что каждая функция-член второго класса яаляется другом для первого. Это не значит, что выполняется обратное: объявление класса в как друга класса А предоставляет в доступ к защищенным членам класса А, однако не предоставляет доступа функциям-членам класса А к защищенным членам класса в.

Поскольку operator+() бинарный (т.е. требует два аргумента), вы видите в объявлении два аргумента для функции (s1 и s2). Функция operator+() берет s1 и складывает его с s2. Результат этого действия возвращается как объект U Dollar.

Унарным оператором, таким как operator++ (), требуется один аргумент. В данном случае operator++() увеличивает поле, в котором содержится сумма в центах. Если эта сумма превышает 100, поле, содержащее сумму в долларах, увеличивается на 1, а поле, содержащее центы, обнуляется,



Нет правила, заставляющего функцию operator+(USDollar&, USDollar&) осуществлять именно сложение. Вы можете заставить функцию operator+() выполнять любые действия; однако выполнение этим оператором чего-либо, кроме сложения, очень-очень плохая идея. Люди привыкли к тому, что их операторы выполняют определенные действия. Вряд ли им понравится, если привычные операторы начнут выполнять непривычные действия.

Оператор += не имеет понятия, как нужно скомбинировать операторы + и =. Таким образом, каждый оператор должен быть перегружен отдельно.



Если вы определили только один оператор— operator++() или operator--(), он будет использован как для префиксной, так и для постфиксной формы. Правда, стандарт C++ не требует от компилятора такой сообразительности, однако большинство компиляторов C++ умеют это делать.



Изначально в C++ не было возможности переопределять префиксный оператор ++x отдельно от его постфиксной версии x++. Однако многим программистам это не нравилось, поэтому правило изменили. В соответствии с этим правилом operator++(ClassName) относится к префиксному оператору, а operator++ (ClassName, int) — к постфиксному. В качестве второго аргумента при этом всегда передается 0. То же правило распространяется и на оператор декремента --.

В работе такие операторы оказываются довольно удобными. Что может быть проще, чем строка d3 = d1 + d2 или ++d3?

А подробнее?

Почему operator+() возвращает сумму по значению, а operator++ () возвращает увеличенный на единицу объект по ссылке? Это не случайность, здесь кроется очень большое отличие между этими операторами!



Мы начинаем осваивать весьма сложную для понимания часть перегрузки операторов, в которой легко запутаться и которую трудно отлаживать.

operator+()

Сложение двух объектов не приводит к изменению ни одного из этих объектов. Таким образом, a + b не изменяет ни a, ни b, а значит, operator+ () не должен сохранять результат сложения в какой-то из этих переменных.

```
// Очень неудачная мысль выполнять сложение так, как это
// сделано в данной программе, поскольку будет изменяться
// значение одного из аргументов
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    s1.cents += s2.cents;
    if (s1.cents >=100)
    {
        s1.cents -= 100;
        s1.dollars++;
    }
    s1.dollars += s2.dollars;
```

```
    return s1;
}
```

Проблема в том, что в результате такого простого присвоения, как u1 = u2 + u3;, будут изменены значения и ul и и2.

Чтобы избежать этого, operator+() должен создавать временный объект, в котором и будет сохранен результат сложения. Поэтому operator+ () конструирует собственный объект, который возвращается этой функцией.

Однако при этом нельзя забывать и еще кое-что! Например, приведенный ниже фрагмент работать не будет.

```
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    unsigned int cents = s1.cents + s2.cents;
    unsigned int dollars = s1.dollars + s2.dollars;
    USDollar result(dollars, cents);
    return result
}
```



Распространенная ошибка № 1. Хотя этот фрагмент откомпилируется без сообщений об ошибке, результат выполнения этой функции будет весьма плачевным. Проблема в том, что возвращается *ссылка* на объект result, который является *локальным* для данной функции. Таким образом, к тому времени, как вызывающая функция сможет использовать возвращаемый результат, объект result уже выйдет из области видимости.

Тогда почему бы нам не выделить блок памяти из кучи так, как это сделано в приведенном ниже примере?

```
USDollar& operator+(USDollar& s1, USDollarS s2)
f
    unsigned int cents = s1.cents + s2.cents;
    unsigned int dollars = s1.dollars + s2.dollars;
    return *new USDollar(dollars, cents);
}
```



Распространенная ошибка № 2, Вы, конечно, можете вернуть ссылку на объект, память под который была выделена из кучи, однако возникнет новая проблема: при этом не предусматривается механизм возврата памяти в кучу. Эта ошибка называется *утечкой памяти*, и ее очень сложно отыскать. Хотя такой оператор и будет работать, он будет потихоньку "истощать" память в куче при каждом выполнении сложения.

Возврат по значению заставляет компилятор создавать временный объект в стеке вызывающей функции. Затем созданный функцией объект копируется в этот временный объект.



Возникает вопрос: как долго существует временный объект, который возвращает operator+ () ? Изначально это не было определено, однако затем создатели стандартов собрались вместе и решили, что такой временный объект остается необходимым до завершения развернутого выражения. Развернутое выражение — это все, что находится перед точкой с запятой. Рассмотрим, например, такой фрагмент:

```
SomeClass f();
LotsClass g0 ;
fn()..
```

```
{
    int  i;
    i = f!)  +  (2-g{)) ;
}
```

Временный объект, возвращенный функцией f(), существует, пока выполняется функция д() и пока выполняется умножение. Там, где стоит точка с запятой, этот объект уже недоступен.

operator++()

В отличие от operator+(), функция operator++() модифицирует свой аргумент. А значит, вам не нужно создавать временный объект или возвращать результат по значению. Вычисляемый результат можно хранить прямо в s. Вызывающей функции может быть возвращен предоставленный оператору аргумент.

```
//это будет отлично работать
USDollar& operator++(USDollar& s)
{
    s.cents++;
    if (s.cents >= 100)
    {
        s.cents -=100;
        s.dollars++;
    }
    return s;
}
```

Рассмотрите приведенный ниже пример, который содержит одну очень хитрую ошибку.

```
//это не очень надежная версия
USDollar operator++(USDollar& s)
{
    s.cents**;
    if (s.cents >= 100)
    {
        s. cents-=100;
        s.dollars++;
    }
    return s;
}
```



Распространенная ошибка № 3. Возвращая s по значению, функция заставляет компилятор генерировать копию объекта. Это отлично сработает в выражениях типа a = ++b, но что будет с выражениями типа ++(++a)? Мы ожидаем, что а будет увеличено на *2.* Однако при приведенном выше переопределении этого оператора, объект а будет увеличен на 1, а затем на 1 будет увеличена его копия, а не сам объект а.

Конструкция вида +* (*+a) не очень распространена, но все же допустима. В любом случае имеется еще множество примеров, в которых такой оператор не будет работать правильно.



Можно сформулировать следующее правило: если оператор изменяет значение своего аргумента, возвращайте аргумент по ссылке. Если оператор не изменяет значения своих аргументов, создавайте новый объект и возвращайте его по значению. Входные аргументы лучше всегда передавать по ссылке.

Операторы как функции-члены

Вместо реализации оператора как внешней функции его можно сделать нестатической функцией-членом. В этом случае класс USDollar будет выглядеть так:

```
class USDollar
{
public:
    USDollar(unsigned int d, unsigned int c);
    USDollar& operator++();
    USDollar operator+(USDollar& s);

protected:
    unsigned int dollars;
    unsigned int cents;
};

USDollar::USDollar(unsigned int d, unsigned int c)
{
    dollars = d;
    cents = c;
    while (cents >= 100)
    {
        dollars++;
        cents -= 100;
    }
}

// operator+ — складывает this и s2 и возвращает
//             результат в новом объекте
USDollar USDollar::operator+(USDollar& s2)
{
    // "this->" необязательно
    unsigned int c = this->cents + s2.cents;
    unsigned int d = this->dollars + s2.dollars;
    USDollar t(d, c);
    return t;
}

// operator++ — увеличивает аргумент,
//              изменяя значение переданного объекта
USDollar& USDollar::operator++()
{
    this->cents++;
    if (this->cents >= 100)
    {
        this->cents -= 100;
        this->dollars++;
    }
    return *this;
}

int main(int argcs, char* pArgs[])
{
    USDollar d1(1, 60);
    USDollar d2(2, 50);
```

```
    USDollar d3 (0, 0) ;
    d3 = d1 + d2;
    ++d3;
    return 0;
}
```

Я специально включил "`this->`" в приведенный ниже пример, чтобы подчеркнуть схожесть между реализациями этого оператора как функции-члена и как внешней функции. Разумеется, это не обязательно, поскольку `this` все равно будет подставляться по умолчанию.

Сравните объявление `USDollar::operator+(USDollar&)` с `::operator+(USDollar&, USDollar&)`. На первый взгляд вариант члена класса имеет на один аргумент меньше, чем глобальный. Приведенный ниже пример сравнивает эти две версии.

```
//operator+ — вариант с функцией — не членом
USDollar operator+ fUSDollars s1, USDollar& s2)
{
    unsigned int c = s1.cents + s2.cents;
    unsigned int d = s1.dollars + s2.dollars;
    USDollar d(dollars,cents);
    return d;
}
//operator+ — вариант с функцией-членом
USDollar operator+ (USDollars s1, USDollar& s2)
{
    unsigned int c = this->cents + s2.cents;
    unsigned int d = this->dollars + s2.dollars;
    USDollar t(dollars,cents);
    return t;
}
```

Как видите, эти функции почти идентичны. Там, где функция — нечлен складывает `s1` с `s2`, функция-член складывает "текущий объект" (тот, на который указывает `this`) с `s2`.



Оператор, который является членом класса, всегда имеет на один аргумент меньше, чем не член класса; при этом подразумевается, что левый аргумент оператора — текущий объект.

Еще одна перегрузка

То, что вы перегрузили один вариант оператора, не значит, что вы перегрузили все операторы C++ прекрасно отличает оператор `operator* (double, USDollar&)` от оператора `operator*(USDollar&, double)`.



Распространенная ошибка № 4. Каждая версия оператора должна быть перегружена отдельно.

Это не настолько существенно связывает нас, как могло бы показаться с первого взгляда. Во-первых, ничто не мешает одному оператору обращаться к другому оператору. В случае с функцией `operator*()` вы можете сделать, например, так:

```
USDollar operator * (double f, USDollar& s)
{
```

```
    //...тело функции здесь...
}
inline USDollar operator* (USDollar& s, double f)
{
    //используем предыдущее определение
    return f*s;
}
```

Вторая версия вызывает первую, просто поставив аргументы в обратном порядке.



Машинный код inline-функции вставляется прямо в место вызова этой функции.

Когда стоит делать операторы членами класса

Возникает вопрос: когда имеет смысл реализовывать оператор как функцию-член, а когда — как внешнюю функцию? Приведенные ниже операторы должны быть реализованы как функции-члены.

```
=  Присвоение
() Вызов функции
[] Индекс
-> Членство в классе
```

Для остальных операторов место их реализации особой роли не играет — в функции-члене или во внешней функции, за небольшим исключением. Например, приведенный ниже оператор не стоит реализовывать как функцию-член.

```
USDollar operator*(double factor, USDollarS s);
void fn(USDollar& principle)
{
    USDollar interestExpense = interest*principle;
    //...
}
```

Чтобы быть функцией-членом, operator* () должен быть членом класса double. Однако простые смертные не могут добавлять операторы во встроенные классы. А значит, такие операторы должны быть внешними функциями.

Если вы имеете доступ к "внутренностям" класса, сделайте перегружаемый оператор членом класса. Это оправдывается, если оператор, например, изменяет объект, с которым осуществляется операция.

Перегрузка операторов с помощью неявного преобразования типов

Есть еще один, принципиально иной способ определения операторов для пользовательских классов. Задумайтесь над приведенным ниже примером.

```
int i = 1;
double d = 1.0;

// выражение №1
d = i + d;
```

```
// выражение №2
i = i + d;
```

Первое выражение складывает int с double. В C++ не определена функция operator+(int, double), но определена функция operator+(double, double). При отсутствии функции (int, double) C++ конвертирует int i в double (говорят, что "i приведено к double"), чтобы использовать версию (double, double). Для обоих выражений выполняется один и тот же процесс, однако со вторым выражением ситуация еще хуже, поскольку результат типа double должен быть урезан до типа переменной i, т.е. int.

Приведение объектов пользовательских типов

Если программист определяет метод приведения из встроенного типа к пользовательскому классу, C++ попытается использовать его для того, чтобы выражение приобрело смысл. Допустим, вы создали конструктор для конвертирования double в USDollar.

```
class USDollar
{
    friend USDollar operator+(USDollar& s1, USDollar& s2);
public:
    USDollar(int d, int c);
    USDollar(double value)
    {
        dollars = (int)value;
        cents = (int)((value - dollars)*100 + 0.5);
    }
    //...то же, что раньше...
}
```

С помощью этого фрагмента кода вы предоставили C++ путь приведения double в USDollar. (Теперь, когда C++ почувствует себя стесненным в наличных средствах, он сможет "заложить" немного double.)

Эту особенность преобразования можно перенести в наши операции:

```
void fn(USDollar& s)
{
  // все приведенные выражения используют
  // operator+ (USDollar&, USDollars)
  s = USDollar(1.5) + s; // явное преобразование...
  s = 1.5 + s;    // ...неявное преобразование..,
  s = s + 1.5;    // ...в обратном порядке...
  s = s + 1;      // даже здесь используется преобразование
                  // int в double, а затем все идет
                  // по старому сценарию
}
```

Теперь вам не нужно определять ни operator+(double, USDollars), ни operator+(USDollar&, double). C++ преобразует double в USDollar и использует уже определенную функцию operator+(USDollars, USDollar&).

Это преобразование может быть указано в явном виде, как это сделано в первом сложении. Однако это же преобразование будет выполнено автоматически, если даже не было указано явно.

Реализация такого преобразования помогает сберечь много усилий за счет уменьшения количества разных операторов, которые в противном случае должен определить программист.



Распространенная ошибка № 5. Довольно опасно доверять C++ автоматическое проведение таких преобразований. Если компилятор не сможет определить, какой тип преобразования нужно использовать, он просто выдаст сообщение об ошибке.

Оператор явного преобразования

Оператор преобразования также может быть перегружен. На практике это выглядит так:

```
class USDollar
{
public:
    USDollar(double value = 0.0);
    operator double()
    {
        return dollars + cents/100.0;
    }
protected:
    unsigned int dollars;
    unsigned int cents;
};
USDollar::USDollar(double value)
{
    dollars = (int)value
    cents = (int)((value - dollars) * 100 + 0.5);
}
```

Оператор преобразования operator double() представляет собой метод преобразования из USDollar в double, в результате которого будет создано действительное значение, равное количеству долларов плюс количество центов, деленное на 100.

На практике операторы такого рода используются следующим образом:

```
int main(int argcs, char* pArgs[])
{
    USDollar d1(2.0), d2(1.5), d3;

    // явно используем оператор преобразования
    d3 = USDollar((double)d1 + (double)d2);

    // или неявно
    d3 = d1 + d2;
    return 0;
}
```

Оператор преобразования состоит из слова operator и следующего за ним названия типа, в который необходимо преобразовать объект. Функция-член USDollar::operator double () обеспечивает механизм преобразования класса USDollar в double. По причинам, которые от меня не зависят, операторы преобразования не имеют типа возвращаемого значения. (Создатели C++ аргументировали это так: "Вам не нужен тип возвращаемого значения, поскольку его можно узнать из имени". К сожалению, создатели C++ не очень последовательны.)

В первом выражении мы преобразуем две переменные USDollar в double, используем для сложения существующую функцию operator+(double, double), а затем преобразуем результат обратно в USDollar с помощью конструктора.

Второе выражение приводит к тому же эффекту, но более хитрым путем. C++ пытается разобраться с выражением d3 = dl + d2, сначала конвертируя d1 и d2 в double, а затем конвертируя сумму обратно в USDollar. Логика этого процесса такая же, как и в первом выражении, однако в данном случае работу, где это только возможно, берет на себя C++.

Этот пример демонстрирует как преимущества, так и недостатки оператора преобразования. Обеспечивая C++ функцией преобразования из USDollar в double, программист освобождается от необходимости создавать полный набор операторов. Класс USDollar может просто воспользоваться операторами, определенными ДЛЯ double.

С другой стороны, предоставление метода преобразования лишает программиста возможности контролировать, какие из операторов определены. Если определена функция преобразования в double, для USDollar будут использоваться операторы double независимо от того, имеет ли это смысл. Кроме того, выполнение большого количества преобразований приведет к значительному уменьшению эффективности программы. Например, приведенное только что простое сложение требует выполнения трех функций преобразования и всех сопутствующих им функций умножения, деления и т.п.

Правила для неявных преобразований

Разбираясь со всеми этими преобразованиями, я так и не объяснил, как C++ поступит со сложением USDollar dl и double d2. Правила для таких операций достаточно просты.

1. Сначала C++ ишет функцию operator+ (USDollar, double).
2. Если она не найдена, C++ ищет функцию, которая может быть использована для преобразования USDollar в double.
3. И наконец, C++ ищет функцию, которая преобразует хоть что-то из этих двух переменных в нечто иное.

Первый пункт ясен, поскольку соответствующая функция уникальна; однако два последних пункта несколько туманны.



Распространенная ошибка № 6. Если возможен более чем один способ выполнения операции, на этапе компиляции будет сгенерирована ошибка.

Если существует два возможных преобразования объекта, компилятор также выдаст ошибку.



Распространенная ошибка № 7. Вы не можете предоставить два способа выполнения преобразования для одного типа. Например, приведенный ниже код вызовет сообщение об ошибке.

```
class A
{
public:
    A(B& b);
};
class B
{
public:
    operator A{);
}
```

Если потребуется выполнить преобразование объекта класса в в объект класса А, компилятор не будет знать, использовать ли ему оператор преобразования B: :operator A() для в или конструктор А: :A(B&) для А, поскольку оба они получают в качестве аргумента объект класса В и создают из него объект класса А.

Возможно, результат преобразования в обоих случаях будет одинаков, но компилятору это не известно. Он должен точно знать, какой из способов преобразования вы имеете в виду. Если это не выясняется однозначно, компилятор умывает свои электронные руки и выдает сообщение об ошибке, предоставляя разбираться в проблеме программисту.

Глава 25

Перегрузка оператора присвоения

В этой главе...

- ✓ Опасная работа, коварная работа, кошмарная работа...
- ✓ Знакомство с перегрузкой оператора присвоения
- ✓ Зашита членов

В главе 24, "Перегрузка операторов", в общих чертах рассматриваются вопросы перегрузки операторов для определенных вами классов. Независимо от того, будете ли вы перегружать все операторы или нет, с перегрузкой оператора присвоения стоит познакомиться как можно раньше.

Вообще говоря, перегрузка операторов в С++ довольно опасное занятие. Однако, если вы будете следовать приведенному в этой главе шаблону, то с перегрузкой оператора присвоения cperator=() никаких проблем не возникает.

Опасная работа, коварная работа, кошма/гноя работа...

В языке С определен только один оператор, который может быть применен к структурированным типам: оператор присвоения. Приведенный ниже пример был бы корректен в С, и его результатом была бы побитовая копия структуры source в destination:

```
void fn()
{
    struct mistrust source, destination;
    destination = source;
}
```

Для обеспечения совместимости с С язык С++ определяет оператор присвоения operator={) по умолчанию для всех пользовательских классов. Этот оператор по умолчанию выполняет почленное копирование для каждого члена класса. И, конечно же, этот оператор может быть перегружен с помощью функции operator=() в любом пользовательском классе.

Оператор присвоения очень похож на конструктор копирования, с которым вы встречались в главе 19, "Копирующий конструктор". При использовании оператор присвоения и конструктор копирования выглядят почти идентично:

```
void fn(MyClass& mc)
{
    MyClass newMC = mc;  // Конструктор копирования
    newMC = mc;          // Оператор присвоения
}
```

Различие заключается в том, что, когда вызывается копирующий конструктор для newMC, самого объекта newMC еше не существует. А когда вызывается оператор присвоения. newMC уже является полноценным объектом класса MyClass.

Копирующий конструктор используется при *создании* объекта, который должен быть копией другого объекта. Оператор присвоения используется, когда объект, стоящий справа от оператора, копируется в *существующий* объект, который находится слева от оператора присвоения. Подобно конструктору копирования, оператор присвоения нужен, когда поверхностного копирования недостаточно.

Знакомство перегрузкой оператора присвоения

Эта процедура аналогична перегрузке любого другого оператора. Например, в приведенной ниже программе оператор присвоения реализован как встроенная функция-член класса Name (не забывайте, что оператор присвоения должен быть функцией-членом класса).

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

class Name
{
public:
    Name()
    {
        pName = (char*)0;
    }
    Name(char *pN)
    {
        copyName(pN);
    }
    Name(Name& s)
    {
        copyName(s.pName);
    }
   ~Name()
    {
        deleteName();
    // оператор присвоения
    Name& operator=(Name& s)
    {
        // Сперва уничтожим существующий объект...
        deleteName();

        //...перед тем, как заменить его новым
        copyName(s.pName);

        // Вернем ссылку на существующий объект
        return *this;
    }

protected:
    // copyName — копирует исходную строку pN в локально
    //            выделенный блок памяти
    void copyName(char *pN)
    {
```

```
        int length = strlen(pN) + 1;
        pName = new char[length];
        strncpy(pName, pN, length);
    }

    // deleteName — освобождение памяти по адресу pName
    void deleteName()
    {
        // если есть этот блок памяти...
        if (pName)
        {
            // ...вернуть его куче...
            delete pName;

            // ...и отметить, что этот указатель
            // более недоступен
            pName = 0;
        }
    }

    // pName указывает ка блок памяти, содержащий
    // имя в виде строки в формате ASCIIZ
    char *pName;
};


int main(int argcs, char* pArgs[])
{
    Name s("Claudette");
    Name t("temporary");
    t = s;      // Здесь вызывается оператор присвоения
    return 0;
}
```

Класс Name хранит имя человека в памяти, которая выделена из кучи конструктором. Конструктор и деструктор Name типичны для класса, содержащего динамический массив памяти.

Оператор присвоения имеет вид `operator=()`. Заметьте, что оператор присвоения как бы состоит из деструктора со следующим за ним копирующим конструктором. Это тоже типично. Разберем операцию присвоения в приведенном примере. Объект t содержит связанное с ним имя (temporary). Выполняя присвоение `t = s`, вы сначала должны вызвать функцию deleteName(), чтобы освободить память, которую занимало предыдущее имя, и только после этого для выделения новой памяти, в которую будет записано новое имя, вы можете вызвать функцию `copyName()`.

В копирующем конструкторе не нужен вызов deleteName(), поскольку в момент вызова конструктора объекта еще не существует, а значит, память для размещения имени еще не выделена.



Обычно оператор присвоения состоит из двух частей. Первая схожа с деструктором в плане освобождения ресурсов, которыми владеет данный объект. Вторая часть схожа с конструктором копирования возможностью выделения новых ресурсов для копирования исходного объекта в целевой.

Глубокая проблема создания мелких копий

Попробуем разобраться, чем же нам не угодило обычное почленное копирование? И в самом деле, поверхностного копирования зачастую бывает вполне достаточно, но, например, в случае с Name это не так.

Класс Name содержит ресурсы, созданные специально для использования в этом объекте, а именно блок памяти, на который указывает pName. Это легко увидеть, посмотрев код конструкторов класса. Каждый из них (кроме конструктора по умолчанию) вызывает функцию copyName().

```
// copyName — копирует исходную строку pN в локально
//              выделенный блок памяти
    voici copyName (char *pN)
    {
        int length = strlen(pN) + 1;
        pName = new char[length];
        strncpy(pName, pN, length);
    }
```

Обратите внимание на то, как функция выделяет место в памяти под строку, равную длине исходной, а затем копирует содержимое pN в созданную строку[19].

При почленном копировании получается два объекта класса Name, указывающих на один и тот же фрагмент в памяти, а потому использование этого метода может легко привести к большим проблемам (например, что произойдет при удалении одного из объектов?).

Оператор присвоения класса Name во избежание утечки памяти сначала должен освободить блок, на который указывает pName (с помощью вызова функции deleteName()), а затем запросить новый фрагмент с помощью функции copyName(). Такой метод и называется глубоким копированием.

Почленный подход к С

Заметьте, что я был очень точен и всегда говорил "почленное копирование". Я избегал упрощенного и неопределенного термина "копирование", поскольку он означает примитивное побитовое копирование.

Смысл почленного копирования не очевиден, пока вы не обратитесь к классам, которые содержат в качестве членов объекты другого класса.

```
class MyClass
{
public:
    Name name;
    int age;
    MyClass(char* pName, int newAge) : name(pName )
    (
        age = newAge;
    }
};
void fn()
{
    MyClass a("Kinsey", 16);
    MyClass b("Christa", 1);
    a = b;
}
```

(Если вы до сих пор незнакомы с синтаксисом инициализации : name(pName), самое время вернуться к главе 18, "Аргументация конструирования".)

19

В принципе в случае, когда новое имя не длиннее старого, мы могли бы обойтись без освобождения памяти и выделения новой, просто перезаписав новое имя на место старого (ценой этого упрощения было бы несколько неэкономное использование памяти программой). — Прим. ред.

Оператор присвоения по умолчанию в данном случае отлично работает. Дело в том, что для копирования члена name используется оператор Name::operator=().

Отсюда правило: хорошего пива... виноват!.. если класс сам по себе не распределяет ресурсы (независимо от того, делают ли это его члены), то нет и необходимости в перегрузке оператора присвоения.

Возврат результата присвоения

Обратите внимание: возвращаемое значение функции operator=() имеет тип Names. Я мог бы сделать этот тип void — C++ не возражал бы против этого, но тогда не работал бы приведенный ниже код.

```
void otherFn(Name&);
void fn(Name& oldN)
{
    Name newN;

    //УЕЬ:, ЭТОТ КОД не работает. . .
    otherFn(newN = oldN);

    //... так же, как и этот...
    Name newerN;
    newerN = newN = oldN;
}
```

Результат присвоения newN = oldN имел бы в этом случае тип void, а значит, никак не мог бы использоваться.



Помните, что результат выполнения оператора присвоения — это значение правого аргумента с типом левого. Таким образом, значение выражения (i = 1) равно 1. Именно благодаря этому выражения типа i = j = 1; полны смысла в C++. Переменной i присваивается результат присвоения j = 1, который равен 1.

Объявление функции operator=() как возвращающей ссылку на текущий объект *this оставляет неизменной это семантическое правило C++.

Вторая деталь, на которую стоит обратить внимание: operator=() должен быть объявлен как функция-член. Оператор присвоения, в отличие от других операторов, не может быть перегружен с помощью функции — не члена класса.



Оператор присвоения должен быть нестатической функцией-членом класса. Интересно, что при этом нет никаких ограничений на специальные операторы типа += или *=; эти операторы могут быть функциями — не членами.

Защита членов

Написать оператор присвоения не очень трудно, тем не менее иногда возможны ситуации, когда по тем или иным соображениям он не нужен. Если вы действительно хотите этого, можете сделать выполнение присвоения невозможным, перегрузив оператор по умолчанию защищенным оператором присвоения, например, так:

```
class Name
{
//...то же, что и раньше...
protected:
```

```
    //оператор присвоения
    Names operator=(Name& s)
    {
        return *this;
    }
};
```

Теперь приведенные ниже операции присвоения оказываются запрещенными.

```
void fn(Name& n)
{
    Name newN;
    newN = n;
    // Ошибка: эта функция не имеет доступа к operator=()!
}
```

Теперь функция fn () не имеет доступа к защищенному оператору присвоения. Такой прием может уберечь вас от проблем, связанных с перегрузкой оператора присвоения и его использованием по умолчанию.

Глава 26

Использование потоков ввода-вывода

В этой главе...

- ✓ Нырнем в поток...
- ✓ Знакомство с подклассами fstream
- ✓ Подклассы strstream
- ✓ Манипулирование манипуляторами
- ✓ Написание собственных операторов вставки
- ✓ Создание "умных" операторов

В главе 11, "Отладка программ на C++", были поверхностно затронуты вопросы, связанные с потоками ввода-вывода. Если вы сопоставите приведенную в этой главе информацию с материалами главы 24, "Перегрузка операторов", то поймете, что потоки ввода-вывода основаны не на каком-то новом специфическом множестве символов << и >>, а на операторах сдвига влево и вправо, перегруженных так, чтобы они выполняли соответственно вывод и ввод. (Если вы не знакомы с перегрузкой операторов, сначала прочитайте главу 24, "Перегрузка операторов".)

В этой главе потоки ввода-вывода описываются более детально. Но должен предупредить вас: это слишком большая тема, чтобы всесторонне осветить ее в одной главе; ей посвящены отдельные книги. К счастью для всех нас, написание подавляющего большинства программ не требует глубоких знаний в области потоков ввода-вывода.

Нырнем в поток...

Операторы, составляющие потоки ввода-вывода, определены в заголовочном файле `iostream.h`, который включает в себя прототипы ряда функций `operator>>()` и `operator<<()`. Коды этих функций находятся в стандартной библиотеке, с которой компонуются ваши программы.

```
// операторы для ввода:
istream& operator>>(istreams source, char* pDest) ;
istream& operator>>(istream& source, int& dest) ;
istream& operator>>(istream& source, chars dest) ;
//. . .и так далее...

// операторы для вывода:
istream& operator<<(ostream& dest, char* pSource);
istream& operator«(ostreams dest, ints source);
istream& operator«(ostreams dest, chars source);
//...и так далее...
```



Время вводить новые термины: в применении к потокам ввода-вывода operator>> () называется *оператором извлечения из потока,* a operator<< () — *оператором вставки в поток.*

Рассмотрим, что случится, если написать следующее:

```
#include <iostream.h>
void fn()
{
    cout << "Меня зовут Стефан\n";
}
```

Сначала C++ определит, что левый аргумент имеет тип ostream, а правый — тип char*. Вооруженный этими знаниями, он найдет прототип функции operator<<(ostream&, char*) в заголовочном файле iostream.h. Затем C++ вызовет функцию вставки в поток для char*, передавая ей строку "Меня зовут Стефан\n" и объект cout в качестве аргументов. Другими словами, он вызовет функцию operator<<(cout, "Меня зовут Стефан\п"). Функция для вставки char' в поток, которая является частью стандартной библиотеки C++, выполнит необходимый вывод.

Но откуда компилятору известно, что cout является объектом класса ostream? Этот и еще несколько глобальных объектов объявлены в файле iostream.h (их список приведен в табл. 26.1). Эти объекты автоматически конструируются при запуске программы, до того как main () получаст управление.

*Таблица **26.1.** Стандартные потоки ввода-вывода*

ОБЬЕКТ	КЛАСС	НАЗНАЧЕНИЕ
cin	istream	Стандартный ввод
cout	ostream	Стандартный вывод
cerr	ostream	Стандартный небуферизованный вывод сообщений об ошибках
clog	ostream	Стандартный буферизованный вывод сообщений об ошибках

Куда делись операторы сдвига

Вы могли бы спросить: "А почему именно операторы сдвига? Почему бы не использовать другой оператор? И вообще, зачем использовать именно перегрузку операторов?". А зачем вы задаете так много вопросов?

Начнем с того, что я не принимал в этом выборе никакого участия. Создатели C++ могли бы остановиться на каком-то стандартном имени функции, например output (), и просто перегрузить эту функцию для выполнения вывода всех встроенных типов. Такой составной вывод мог бы выглядеть приблизительно так:

```
void displayName(char* pName, int age)
{
    output(cout, "Имя ");
    output(cout, pName);
    output(cout, "; возраст ") ;
    output(cout, age);
    output(cout, "\n");
}
```

Вместо этого был выбран оператор сдвига влево. Во-первых, это бинарный оператор, т.е. вы можете поставить объект типа ostream слева от него, а выводимый

объект — справа. Во-вторых, оператор сдвига влево имеет очень низкий приоритет. Благодаря этому выражения с ним работают так, как ожидается.

```
#include <iostream.h>
void fnfint a, int b)
{
    cout « "a + b" « a + b « "\n";
    // cperator+ имеет более высокий приоритет,
    // чем operator<<
    // Следовательно, данное выражение
    // интерпретируется как
    //cout « "a + b" « (a + Ь) << "\п";
    //а не как
    //cout « ("a + b" « a) + (b « "\п");
}
```

В-третьих, оператор сдвига влево вычисляется слева направо, что позволяет записывать длинные выражения вывода. Так, приведенное выше выражение интерпретируется следующим образом:

```
#include <iostream.h>
void fn(int a, int b) f
  ((cout « "a + b") « a + b) « "\n";
}
```

Но, на мой взгляд, настоящая причина кроется в том, что оператор сдвига здорово смотрится. Символы "<<" выглядят так, как будто что-то выводится из программы, а " » " — как будто что-то вносится в нее. И наконец, "пуркуа бы и не па"?..

А что же такое ostream? Объект класса ostream содержит члены, необходимые для управления потоком ввода-вывода. А istream соответственно описывает поток ввода.

В С этому эквивалентна структура FILE, определенная в stdio.h. Функция fopen () открывает файл для ввода и вывода и возвращает указатель на объект типа FILE, в котором хранится информация, необходимая для осуществления операций ввода-вывода. Этот объект используется при вызове функций f...(), таких как fprintf(), fscanf() и fgets().

В составе библиотеки потоков ввода-вывода определено еше несколько подклассов ostream и istream, которые используются для ввода и вывода в файлы и внутренние буферы.

Знакомство с подклассами fstream

Подклассы ofstream, ifstream и fstream объявлены в заголовочном файле fstream.h и обеспечивают потоки ввода-вывода в дисковые файлы. Эти три класса предоставляют множество функций для управления вводом и выводом, многие из которых наследуются от ostream и istream. Полный список этих функций вы можете найти в документации к компилятору, а здесь я приведу только несколько из них, чтобы вы могли с чего-то начать.

Класс ofstream, который используется для файлового вывода, имеет несколько конструкторов; наиболее часто применяется следующий:

```
ofstream::ofstream(char* pFileName,
                   int mode = ios::out,
                   int prot = filebuff::openprot);
```

Первый аргумент этого конструктора — указатель на имя открываемого файла. Второй и третий аргументы определяют, как именно должен быть открыт файл. Корректные значения аргумента mode приведены в табл. 26.2, а prot — в табл. 26.3. Эти значения являются битовыми полями, к которым применяется оператор OR (классы ios и filebuff— родительские по отношению к ostream).

Таблица 26.2. Значения аргумента mode в конструкторе класса ofstream

ФЛАГ	НАЗНАЧЕНИЕ
ios::ate	Дописывать в конец файла, если он существует
ios::in	Открыть файл для ввода (подразумевается для istream)
ios::out	Открыть файл для вывода (подразумевается для ostream)
ios::trunc	Обрезать файл до нулевой длины, если он существует (используется по умолчанию)
ios::nocreate	Если файла не существует, вернуть сообщение об ошибке
ios::noreplace	Если файл существует, вернуть сообщение об ошибке
ios::binary	Открыть файл в бинарном режиме (альтернатива текстовому режиму)

Таблица 26.3. Значения аргумента prot в конструкторе класса ofstream

ФЛАГ	НАЗНАЧЕНИЕ
filebuf::openprot	Режим совместного чтения и записи
filebuf::sh_none	Исключительный режим без совместного доступа
filebuf::sh_read	Режим совместного чтения
filebuf::sh_write	Режим совместной записи

Приведенная ниже программа открывает файл MYNAME, а затем записывает в него некоторую важную информацию.

```
#include <fstream.h>

void fn()
{
    // Откроем текстовый файл MYNAME для записи,
    // уничтожив имевшееся в нем содержимое
    ofstream myn("MYNAME");

    //теперь запишем в файл
    туп << "У попа была собака\п";
}
```

Конструктор ofstream::ofstream(char*) получает только имя, а потому использует для режима открытия файла значения по умолчанию. Если файл MYNAME уже существует, он урезается; в противном случае создается новый файл MYNAME. Кроме того, файл открывается в режиме совместного чтения и записи.

Второй конструктор, ofstream::ofstream(char*, int), позволяет программисту указывать другие режимы ввода-вывода. Например, если бы я захотел открыть файл в бинарном режиме и произвести запись в конец этого файла (если он уже существует), я мог бы создать объект класса ofstream так, как приведено ниже (напомню, что в бинарном режиме при выводе не выполняется преобразование символа новой строки \п в пару символов перевода каретки и новой строки \r\n, так же как при вводе не происходит обратного преобразования).

```
#include <fstream.h>

void fn()
{
    //откроем бинарный файл BINFILE для записи; если он
    //существует, дописываем информацию в конец файла
    ofstream bfile("BINFILE", ios::binary|ios::ate);
    //..продолжение программы...
}
```

Функция-член bad () возвращает 1, если при работе с файловым объектом возникли ошибки. В приведенном выше примере для проверки того, что файл открыт правильно, надо было написать следующее:

```
#include <fstream.h>

void fn()
{
    ofstream myn("MYNAME");
    if(myn.bad())    //Если открыть файл не удалось...
    {
        cerr << "Ошибка при открытии файла MYNAME\n";
        return;
    }
    //Теперь произведем запись в файл
    туп << "У попа была собака\п";
}
```



Все попытки обратиться к объекту класса ofstream, который содержит ошибку, не вызовут никакого действия, пока флаг ошибки не будет сброшен с помощью функции clear().

Деструктор класса ofstream автоматически закрывает файл. В предыдущем примере файл был закрыт при выходе из функции.

Класс ifstream работает для ввода почти так же, как ofstream для вывода, что и демонстрирует приведенный ниже пример.

```
#include <fstream.h>

void fr. ()
(
    // Откроем файл для чтения; не создавать
    // файл, если он отсутствует
    ifstream bankStatement("STATEMNT", ios::nocreate);
    if (bankStatement.bad())
    {
        cerr « "Не могу найти файл STATEMNT\n";
        return;
    }

    // Цикл считывания из файла, пока
    // не будет достигнут его конец (eof)
    while (!bankStatement.eof())
    {
        bankStatement » accountNumber >> amount;
        //... обработаем полученную запись...
    }
}
```

Функция открывает файл STATEMNT, конструируя объект bankStatement. Если такого файла не существует, этот объект не создается (ведь если вы ожидаете получить какую-то информацию из этого файла, бессмысленно создавать новый пустой файл). Если объект неправилен (например, файл не существовал), функция выводит сообщение об ошибке и завершается. В противном случае функция выполняет считывание пар переменных accountNumber и withdrawal, пока файл не закончится (bankstatement.eof() == true).



Попытка прочитать информацию с помощью объекта класса ifstream с установленным флагом ошибки приведет к немедленному возврату без считывания чего-либо. Для сброса флага ошибки используйте функцию clear().

Класс fstream похож на комбинацию классов ifstream и ofstream (кстати, он и наследуется от обоих этих классов). Объект класса fstream может быть создан как для ввода, так и для вывода.

Подклассы strstream

Классы istrstream, ostrstream и strstream определены в заголовочном файле с именем strstrea.h или strstream.h.



Операционная система MS DOS ограничивала имена файлов восьмью символами с трехбуквенным расширением; другими словами, использовала имена в формате 8.3 DOS. Это ограничение ушло в историю, однако многие компиляторы, включая Visual C++, придерживаются совместимости с именами типа 8.3 и используют имя strstrea.h. Компилятор GNU C++ использует полное имя strstream.h.

Классы из strstream.h позволяют использовать операции, определенные для файлов в классах fstream, для строк в памяти. Это очень похоже на функции sprintf() и sscanf() в С.

Приведенный ниже фрагмент кода использует поток ввода для разбора передаваемой функции строки.

```
//Если это компилятор Visual C++ или подобный
#ifdef _WIN32
//...использовать имя типа 8.3...
#include <strstrea.h>
//...в противном случае...
#else
//использовать полное имя
#include <strstream.h>
#endif
//parseString — демонстрирует классы строчных потоков,
//              считывая переданный буфер, как если бы
//              это был простой файл
char* parseString(char* pString)
{
    istrstream inp(pString, 0);

    int accountNumber;
    float balance;
    inp >> accountNumber >> balance;
    char* pBuffer = new char[128];
```

```
    ostrstream out(pBuffer, 128);

    out << "Номер счета = " << accountNumber
        << ", баланс = $" << balance;
    return pBuffer;
}
```

Пусть для определенности pstring указывает на строку "1234 100.0".

Объект inp связывается со строкой с помощью конструктора istrstream. Второй аргумент конструктора — длина строки. В нашем примере этот аргумент равен 0, что означает "читать, пока не будет достигнут конец строки".

Объект out для вывода связан с буфером, на который указывает pBuffer. Второй аргумент конструктора в этом случае также представляет собой размер буфера. Третий аргумент относится к режиму работы потока, который по умолчанию равен ios::out. Конечно же, вы можете установить этот аргумент равным ios::ate, если хотите дописывать информацию в буфер, вместо того чтобы переписать его.

Фрагмент кода, отвечающий за ввод, записывает значение 1234 в переменную accountNumber и значение 100 в переменную balance. Фрагмент, отвечающий за вывод, дописывает строку "Номер счета =" в буфер вывода *pBuffer, за которым следует число 1234 из переменной accountNumber, и т.д.

В результате выполнения этого кода для данного ввода буфер вывода будет содержать строку

"Номер счета = 1234, баланс = $100.00"

Директива #ifdef в начале предыдущего фрагмента кода нужна для того, чтобы подключить правильный заголовочный файл. _WIN32 всегда определена, когда программа компилируется Visual C++, и не определена, если программа компилируется GNU C++. Таким образом, когда программа компилируется с помощью Visual C++, включается файл strstrea.h; в противном случае используется файл strstream.h.

Еще раз рассмотрите этот фрагмент кода. Если бы потребовалось, чтобы inp и out использовали файлы, в программе следовало бы изменить только конструкторы. В остальном программа была бы неизменной. Работа с буфером памяти, по сути, ничем не отличается от работы с внешним файлом.

Манипулирование манипуляторами

Обычно потоки ввода-вывода для выведения чисел и символов используют формат по умолчанию, который оказывается вполне подходящим для решения большинства задач.

Однако мне страшно не понравилось, когда общая сумма в моей любимой программе BUDGET была выведена как 249.600006 вместо ожидаемого 249.6 (а еще лучше — 249.60). Необходимо каким-то образом указать программе количество выводимых цифр после десятичной точки. И такой способ есть; более того, в C++ он не единственный.



В зависимости от установок по умолчанию вашего компилятора, вы можете увидеть на экране 249.6. Однако хотелось бы добиться того, чтобы выводилось именно 249.60.

Во-первых, форматом можно управлять с помощью серии функций-членов объекта потока. Например, количество разрядов для отображения можно установить, используя функцию precision():

```
#include <iostream.h>
void fn(float interest, float dollarAmount)
```

```
{
    cout << "Сумма в долларах = ";
    cout.precision(2);
    cout << dollarAmount;
    cout.precision(4);
    cout << interest
         << "\n";
}
```

В этом примере с помощью функции precision (> вывод значения dollarAmount устанавливается с точностью двух знаков после запятой. Благодаря этому вы можете увидеть на экране число 249.60 — именно то, что вы хотели. Затем устанавливается вывод процентов с точностью четырех знаков после запятой.

Второй путь связан с использованием так называемых манипуляторов. (Звучит страшновато, не так ли?) Манипуляторы — это объекты, определенные в заголовочном файле iomanip.h, которые приводят к тому же эффекту, что и описанные выше функции-члены (чтобы иметь возможность пользоваться манипуляторами, вы должны не забыть включить iomanip.h в программу). Единственное преимущество манипуляторов в том, что программа может включать их прямо в поток, не прибегая к вызову отдельной функции.

Если вы перепишете предыдущий пример так, чтобы в нем использовались манипуляторы, программа будет иметь следующий вид:

```
#include <iostream.h>
#include <iomanip.h>
void fn(float interest, float dollarAmount)
{
    cout << "Сумма в долларах = ";
         << setprecision(2) << dollarAmount
         << setprecision(4) « interest
         << "\n";
}
```

Наиболее распространенные манипуляторы и их назначение приведены в табл. 26.4.

Таблица 26.4. Основные манипуляторы и функции управления форматом потока

МАНИПУЛЯТОР	ФУНКЦИЯ-ЧЛЕН	ОПИСАНИЕ
dec	flags (10)	Перейти в десятичную систему счисления
hex	flags(16)	Перейти в шестнадцатеричную систему счисления
oct	flags (8)	Перейти в восьмеричную систему счисления
setfill(c)	fill(c)	Установить символ заполнения с
setprecision(c)	precision(c)	Установить количество отображаемых знаков после запятой в с
setw(n)	width(n)	Установить ширину поля равной п символов*

*Это значение воздействует на вывод одного поля и возвращается к значению по умолчанию.



Внимательно следите за параметром ширины поля (функция width (n) либо манипулятор setw(c)). Большинство параметров сохраняют свое значение до тех пор, пока оно не будет изменено новым вызовом, однако для параметра ширины поля это не так. Этот параметр возвращается к значению по умолчанию, как только будет выполнен следующий вывод в поток. Например, не надейтесь, что приведенный ниже фрагмент кода выведет два целочисленных значения длиной в 8 символов.

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
    cout. << setw(8)  //ширина поля равна 8...
          << 10       //...для 10, но ...
          << 20       // для 20 равна значению по умолчанию
          «  "\n";
}
```

В результате выполнения этого кода сначала будет выведено восьмисимвольное целое число, а за ним — двухсимвольное. Для вывода двух восьмисимвольных значений нужно сделать так:

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
    cout; << setw(8)  //установить ширину. . .
          «  10
          «  setw(8)
          << 20       //...обновить ее
          «  "\n";
}
```

Таким образом, если вам нужно вывести несколько значений, но вас не устраивает длина поля по умолчанию, для каждого значения необходимо включать в вывод манипулятор setw().

Какой же метод лучше — с использованием манипуляторов или функций? Функции-члены предоставляют больше контроля над свойствами потока — хотя бы потому, что их больше. Кроме того, функции-члены обязательно возвращают предыдущее значение изменяемого параметра, так что вы всегда имеете возможность восстановить прежнее значение параметра. И наконец, существуют версии этих функций, позволяющие узнать текущее значение параметра, не изменяя его. Использование этой возможности показано в приведенном ниже примере.

```
#include <iostream.h>
void fn(float value)
{
    int previousPrecision;
    // Вы можете узнать текущую точность так:
    previousPrecision = cout.precision();

    // Можно также сохранить старое значение,
    // одновременно изменяя его на новое
    previousPrecision = cout.precision(2);
    cout << value;

    // Восстановим предыдущее значение
    cout.precision(previousPrecision);
}
```

Несмотря на все преимущества "функционального" подхода, манипуляторы более распространены; возможно, просто потому, что они "круче" выглядят. Используйте то, что вам больше нравится, но в чужом коде будьте готовы увидеть оба варианта.

Написание собственных операторов вставки

С++ обладает потрясающей способностью перегружать оператор сдвига для выполнения вывода. Это значит, что вы тоже можете перегрузить этот оператор, чтобы осуществить вывод собственных классов.

Это наиболее важная особенность потоков ввода-вывода, к описанию которой я шел на протяжении всей этой главы, но до настоящего момента старательно избегал упоминания о ней. Итак, представьте себе, что класс USDollar из главы 24, "Перегрузка операторов", дополнен функцией display!).

```
#ifdef _WIN32
#include <strstrea.h>
#else
#include <strstream.h>
#endif

#include <iomanip.h>
class USDollar
{
public:
    USDollar(double v = 0.0)
    {
        // Удалим дробную часть
        dollars =  (int)v;

        // Преобразуем дробную часть в центы,
        //добавив 0.5 для округления
        cents = int((v - dollars) *· 100.0 + 0.5) ;
    }

    operator double()
    {
        return dollars + cents / 100.0;
    }

    void display(ostream& out)
    {
        out << '$' << dollars « '.'
        // установим заполнение нулем для центов
        << setfill('0') << setw(2) « cents
        // вернем настройку заполнения пробелами
        <<setfill(' ') ;
    }

protected:
    unsigned int dollars;
    unsigned int cents;
};

// operator« — оператор вставки в поток нашего класса
ostream& operator << (ostream& o, USDollar& d)
{
    d.display(o);
    return o;
}
```

```
int main(int argcs, char* pArgs[])
{
    USDollar usd(1.50);
    cout << "Начальная сумма = " << usd << "\n";
    usd = 2.0 * usd;
    cout << "Теперь сумка равна " << usd << "\n";
    return 0;
}
```

Функция display() начинается с вывода символа $, после чего выводится сумма с обязательной десятичной точкой. Обратите внимание, что вывод выполняется в любой потоковый объект ostream, а не только в cout. Это позволяет использовать функцию display() и для объектов fstream и strstream, которые являются подклассами класса ostream.

Когда приходит время отображать количество центов, функция display () устанавливает ширину поля в два символа и заполнение символом 0. Благодаря этому можно быть уверенным, что числа меньше 10 будут отображены правильно. Обратите внимание, что вместо прямого обращения к функции display() класс USDollar определяет operator<<(ostream&, USDollar&). Теперь программист может выводить объекты класса uSDollar с такой же непринужденностью и изяществом, как и объекты встроенных типов, что и демонстрирует функция main ().

В результате выполнения этой программы будут выведены следующие сообщения:

```
Начальная сумма = $1.50
Теперь сумма равна $3.00
```

Вы можете поинтересоваться, зачем функция cperator<<() возвращает объект класса ostream, передаваемый ей в качестве аргумента. Дело в том, что это позволяет связывать один оператор с другими операторами в одном и том же выражении. Поскольку operator<< () обрабатывается слева направо, выражение

```
void fn(USDollar& usd, float i)
{
    cout << "Сумма " << usd << ", проценты = " << i;
}
```

интерпретируется как

```
void fn(USDollar& usd, float i)
{
     (((cout << "Сумма ") << usd) << ", проценты = ") << i;
}
```

Первый оператор вставки в поток выводит строку "Сумма" в cout. Результатом этого выражения является объект cout, который затем передается функции operator<<(ostream&, USDollar&). Поскольку этот оператор возвращает свой объект как ссылку на ostream, данный объект может быть передан следующему в очереди оператору вставки в поток.

Если бы типом возвращаемого значения оператора вставки в поток вы объявили void, оператор бы оставался работоспособным, но в приведенном примере вызывал бы ошибку компиляции, поскольку вставить строку в void нельзя. Приведенная же в следующем примере ошибка еще опаснее, поскольку ее трудно обнаружить.

```
ostream& operator<<(ostreams os, USDollars usd)
{
    usd.display(os);
    return cout;
}
```

Заметьте, что эта функция возвращает не объект типа ostream, который она получила, а объект cout, который имеет тот же тип ostream. Тут очень легко ошибиться, поскольку cout является наиболее часто используемым объектом типа ostream. Эта ошибка не проявится, пока не будет использована приведенная ниже конструкция.

```
void storeAccount(int account,
                  USDollar balance,
                  char* pName)
{
    ofstream outFile("ACCOUNTS", ios::ate);
    outFile << account << balance << pName;
}
```

Целочисленная переменная account выводится в outFile с помощью функции operator<<(ostream&, int&), которая возвращает outFile. Затем в outFile с помощью функции operator<<(ostream&, USDollarS) выводится USDollar. Эта функция возвращает неправильный объект — cout вместо outFile. В результате pName выводится в cout вместо вывода в файл, как требуется.

Создание "умных" операторов

Наверняка вам захочется, чтобы операторы вставки в поток были настолько умны, что вы могли бы сказать им: cout << baseClassObject, а уж затем С++ сам бы выбирал оператор вставки в поток необходимого подкласса, так же как он выбирает необходимую виртуальную функцию. Увы, поскольку оператор вставки в поток не является функцией-членом, вы не можете объявить его виртуальным. Однако для умного программиста это не проблема, что и демонстрирует приведенный ниже пример.

```
#include <iostream.h>
#include <iomanip.h>
class Currency
i
public:
    Currency(double v = 0.0)
    {
        // Выделяем целую часть
        unit = (int)v;
        // Округляем дробную часть
        cents = int((v - unit)* 100.0 + 0.5);
    }

    virtual void display(ostream& out) = 0;

protected:
    unsigned int unit;
    unsigned int cents;
};

class USDollar : public Currency
{
  public:
    USDollar(double v = 0.0) : Currency(v)
    {
    }
```

```
    // формат отображения: $123.45
    virtual void display(ostream& out)
    {
        cut << '$' « unit << '.'
            << setfill('0') << setw(2) << cents
            << setfill(' ');
    }
};

class UAH : public Currency
(
  public:
    UAH(couble v = 0.0) : Currency(v)
    {
    }

    // формат отображения: 123.00 грн.
    virtual void display(ostream& out)
    {
      out. << unit << ' . '
        // Установим заполнение 0 для копеек
        << setfill('O') << setw(2) << cents
        // Вернем заполнение пробелами
        << setfill(' ')
        « " грн.";
    }
};
ostream& operator<< (ostream& o, Currency& c)
{
    c.display(o);
    return o;
}
void fn (Currency& c)
{
    // Используемый ниже вывод полиморфен, поскольку
    // operator (ostream&, Currency&) работает с
    // виртуальной функцией-членом
    cout << "Сумма равна " << c
         « "\п";
}

int main(int argcs, char* pArgs[])
{
    //создать USDollar и вывести его, используя
    //соответствующий доллару формат
    USDollar usd(1.50);
    fn(usd);

    //теперь создать гривну и вывести ее
    UAH d(3.00);
    fn(d);
    return 0;
}
```

Класс Currency имеет два подкласса — USDollar и UAH. В классе Currency функция display () объявлена чисто виртуальной. В обоих подклассах эта функция перегружается с тем, чтобы выводить объект в корректном виде. Вызов display () из функции operator<<() является виртуальным, и, когда функции operator<<() пе-

редается объект USDollar, эта функция выводит объект как доллар. Если же был передан объект класса UAH, operator<< () выводит его как гривну.

Итак, хотя функция operator<<() и не виртуальна, то, что она вызывает виртуальную функцию, приводит к корректным результатам:

```
Сумма разна $1.50
Сумма равна 3.00 грн.
```

Это еще одна из причин, по которой я предпочитаю перекладывать работу, связанную с выводом на функцию-член, заставляя оператор (который не является функцией-членом) обращаться к display ().

Глава 27

Обработка ошибок и исключения

В этой главе...

- ✓ Зачем нужен новый механизм обработки ошибок
- ✓ Механизм исключительных ситуаций
- ✓ Так что же мы будем бросать?

Я знаю, как трудно с этим смириться, но факт остается фактом: иногда функции работают неправильно. И не только мои. Традиционно вызывающей программе сообщается об ошибке посредством возвращаемого функцией значения. Однако язык С++ предоставляет новый, улучшенный механизм выявления и обработки ошибок с помощью *исключительных ситуаций*, или *исключений* (exceptions). Исключение — это отступление от общего правила, т.е. случай, когда то или иное правило либо принцип неприменимы. Можно дать и такое определение: исключение — это неожиданное (и, надо полагать, нежелательное) состояние, которое возникает во время выполнения программы.

Механизм исключительных ситуаций базируется на ключевых словах `try` (попытаться), `throw` (бросить) и `catch` (поймать). В общих чертах этот механизм работает так: функция *пытается* выполнить фрагмент кода. Если в коде содержится ошибка, она *бросает* (генерирует) сообщение об ошибке, которое должна *поймать* (перехватить) вызывающая функция.

Это продемонстрировано в приведенном ниже фрагменте.

```
#include <iostream.h>

//factorial — вычисляет факториал
int factorial(int n)
{
    // Поскольку функция не может работать с
    // отрицательными n, мы сразу проверяем
    // допустимость переданного аргумента
    if (n < 0)
    {
        throw "Отрицательный аргумент";
    }

    // теперь вычислим факториал
    int accum = 1;
    while(n > 0)
    {
        accum *= n;
        n--;
    }
    return accum;
}

int main(int argcs, char* pArgs[])
{
```

```
    try {
        // Эта строка генерирует исключение
        cout << "Factorial of -1 is "
             << factorial(-1) << endl;

        // Управление никогда не дойдет до этой строки
        cout << "Factorial of 10 is "
             << factorial(10) << endl;
    }

    // Управление будет передано сюда
    catch(char* pError) {
        cout << "Возникла ошибка: " << pError << endl;
    }
    return 0;
}
```

Функция main() начинается с блока, выделенного ключевым словом try. В этом блоке выполнение кода ничем не отличается от выполнения вне блока. В данном случае main() пытается вычислить факториал отрицательного числа. Однако функцию factorial() не так легко одурачить, поскольку она достаточно умно написана и обнаруживает, что запрос некорректен. При этом она генерирует сообщение об ошибке с помощью ключевого слова throw. Управление передается фрагменту, находящемуся сразу за закрывающей фигурной скобкой блока try и отвечающему за перехват сообщения об ошибке.

Зачем нужен новый механизм обработки ошибок

Что плохого в методе возврата ошибки, подобном применяемому в FORTRAN? Факториал не может быть отрицательным, поэтому я мог бы сказать что-то вроде: "Если функция factorial() обнаруживает ошибку, она возвращает отрицательное число. Значение отрицательного числа будет указывать на источник проблемы". Чем же плох такой метод? Ведь так было всегда.

К сожалению, здесь возникает несколько проблем. Во-первых, хотя результат факториала не может быть отрицательным, другим функциям повезло гораздо меньше. Например, вы не можете взять логарифм от отрицательного числа, но сам логарифм может быть как отрицательным, так и положительным, а поэтому возврат отрицательного числа не обязательно будет означать ошибку.

Во-вторых, в целочисленной переменной не передашь много информации. Можно, конечно, обозначить ситуацию "аргумент отрицательный" как -1, ситуацию "аргумент слишком большой" как -2 и т.д. Но если аргумент слишком большой, я хотел бы знать, какой именно, поскольку это поможет мне найти источник проблемы. Однако в целочисленной переменной такую информацию не сохранишь.

В-третьих, проверка возвращаемого значения вовсе не обязательна. Чтобы понять, что я имею в виду, представьте себе, что кто-то написал функцию factorial(), которая послушно проверяет, находится ли ее аргумент в допустимых границах. Однако, если код, вызвавший эту функцию, не будет проверять возвращаемое значение, это не приведет ни к чему хорошему. Конечно, я мог бы ввести в функцию всякие страшные угрозы наподобие "Вы обязательно должны проверить сообщения об ошибках, иначе...", но, думаю, не стоит объяснять, что язык не может никого ни к чему принудить.

Даже если моя функция проверяет наличие ошибки в factorial () или любой другой функции, что она может с ней сделать? Пожалуй, только вывести сообщение об ошибке (которое я сам написал) и вернуть другое значение, указывающее на наличие ошибки, вызывающей функции, которая, скорее всего, повторит весь этот процесс. В результате программа будет переполнена кодом, подобным приведенному ниже.

```
errRtn = someFunc();
if (errRtn)
{
    errorOut("Ошибка при вызове someFn()");
    return MY_ERROR_1
}
errRtn = someOtherFunc() ;
if (errRtn)
{
    errorOut("Ошибка при вызове someOtherFn()");
    return MY_ERROR_1
}
```

Такой механизм имеет ряд недостатков.

- ✓ Изобилует повторениями.
- ✓ Заставляет программиста отслеживать множество разных ошибок и писать код для обработки всех возможных вариантов.
- ✓ Смешивает код, отвечающий за обработку ошибок, с обычным кодом, что не добавляет ясности программе...

Эти недостатки выглядят не очень страшно в простом примере, но могут превратиться в большие проблемы, когда программа станет более сложной. В результате такой подход приводит к тому, что обработкой ошибок занимается 90% кода.

Механизм исключительных ситуаций позволяет обойти эти проблемы, отделяя код обработки ошибок от обычного кода. Кроме того, наличие исключений делает обработку ошибок обязательной. Если ваша функция не обрабатывает сгенерированное исключение, управление передается далее по цепочке вызывающих функций, пока C++ не найдет функцию, которая обработает возникшую проблему. Это также дает возможность игнорировать ошибки, которые вы не в состоянии обработать.

Механизм исключительных ситуаций

Познакомимся поближе с тем, как программа обрабатывает исключительную ситуацию. При возникновении исключения (throw) C++ первым делом копирует сгенерированный объект в некоторое нейтральное место. После этого просматривается конец текущего блока try.

Если блок try в данной функции не найден, управление передается вызывающей функции, где и осуществляется поиск обработчика. Если и здесь не найден блок try, процесс повторяется далее, вверх по стеку вызывающих функций. Этот процесс называется *разворачиванием стека*.

Важной особенностью разворачивания стека является то, что на каждом его этапе все объекты, которые выходят из области видимости, уничтожаются так же, как если бы функция выполнила команду return. Это оберегает программу от потери ресурсов и праздно шатающихся неуничтоженных объектов.

Когда необходимый блок try найден, программа ищет первый блок catch (который должен находиться сразу за закрывающей скобкой блока try). Если тип сгенерированного объекта совпадает с типом аргумента, указанным в блоке catch,

управление перелается этому блоку; если же нет, проверяется следующий блок catch. Если в результате подходящий блок не найден, программа продолжает поиск уровнем выше, пока не будет обнаружен необходимый блок catch. Если искомый блок так и не найден, программа аварийно завершается.

Рассмотрим приведенный ниже пример.

```
#include <iostream.h>
#include <stdio.h>

class Obj
{
public:
    Obj (char c)
    {
        label = c;
        cout << "Конструируем объект " << label << endl;
    }
    ~Obj ()
    {
        cout << "Ликвидируем объект " << label << endl;
    }

protected:
    char label;
};

void f1() ;
void f2();
int main(int, char*[])
{
    Obj a('a');
    try
    {
        Obj b('b') ;
        f1 ();
    }
    catch(float f)
    {
        cout << "Исключение float" << endl;
    }
    catch(int i)
    {
        cout << "Исключение int" << endl;
    }
    catch (...)
    {
        cout << "Исключение..." << endl;
    }
    return 0;
}

void f1 ()
{
    try
    {
        Obj c('c');
        f2() ;
    }
    catch(char* pMsg)
```

```
    {
        cout << " Исключение char*" << endl;
    }
}
void f2()
{
    Obj d('d');
    throw 10;
1
```

В результате работы этой программы на экран будет выведен следующий текст:

```
Конструируем объект a
Конструируем объект Ь
Конструируем объект c
Конструируем объект d
Ликвидируем объект d
Ликвидируем объект c
Ликвидируем объект b
Исключение int
Ликвидируем объект a
```

Как видите, прежде чем в функции f2 () происходит исключение int, конструируются четыре объекта — a, b, c и d. Поскольку в f2() блок try не определен, С++ разворачивает стек вызовов функций, что приводит к ликвидации объекта d при разворачивании стека f2 (). В функции fl() определен блок try, но его блок catch воспринимает только char*, что не совпадает с брошенным объектом int. Поэтому С++ продолжает просмотр, что приводит к разворачиванию стека функции f1() (при этом ликвидируется объект c).

В функции main{) С++ находит еще один блок try. Выход из этого блока приводит к выходу из области видимости объекта Ь. Первый за блоком try блок catch принимает float, что вновь не совпадает с нашим int, поэтому пропускается и этот блок. Однако следующий блок catch наконец-то воспринимает int, и управление переходит к нему. Последний блок catch, который воспринимает любой объект, пропускается, поскольку необходимый блок catch уже найден и исключение обработано.

Жак что же мы будем бросать?

За ключевым словом throw следует выражение, которое создает объект некоторого типа. В приведенных здесь примерах мы генерировали переменные типа int, но на самом деле ключевое слово throw работает с любым типом объекта. Это значит, что вы можете "бросать" любое количество информации. Рассмотрим приведенное ниже определение класса.

```
#include <iostream.h>
#include <string.h>

//Exception — универсальный класс
//обработки исключительных ситуаций
class Exception
{
public:
    Exception(char* pMsg, char* pFile, int nLine)
    {
        strncpy(msg, pMsg, sizeof msg);
        msg[sizeof msg — 1] = '\0 ';
        strncpy(file, pFile, sizeof file);
```

```
        file[sizeof file - 1] = '\0';
        lineNum = nLine;
    }
    virtual void display(ostream& out)
    {
        out << "Ошибка < " << msg << ">\n";
        out << "обнаружена в строке #" << lineNum
            << ", файла" << file << endl;
    }
protected:
    //сообщение об ошибке
    char msg[80];

    //имя файла и строка, в которой возникла ошибка
    char file[80];
    int lineNum;
};
```

Генерация исключения при этом будет выглядеть следующим образом:

```
throw Exception("Отрицательный аргумент факториала",
                __LINE__,__FILE__);
```



Являясь встроенными макроопределениями, __FILE__ и __LINE__ представляют собой имя исходного файла и текущую строку в нем.



Класс ostream, использованный в функции display (), является базовым для потоков вывода (см. главу 26, "Использование потоков ввода-вывода").

Соответствующий блок catch выглядит довольно просто:

```
void myFunc()
{
    try
    {
        //...любой вызов
    }
    //захват объекта Exception
    catch(Exception& x)
    {
        //используем встроенную функцию-член
        x.display(cerr);
    }
}
```

Блок catch перехватывает исключение Exception, а затем использует встроенную функцию-член display () для отображения сообщения об ошибке.



Объект cerr представляет собой поток вывода для сообщений об ошибках, т.е. почти то же, что и cout. Разница между cout и cerr существенна только для профессиональных программистов.

Класс Exception является универсальным для сообщений об ошибках. Этот класс, конечно, может быть расширен другими подклассами. Например, я могу определить класс InvalidArgumentException для хранения значения некорректного аргумента в дополнение к сообщению об ошибке.

```
class InvalidArgumentException : public Exception
{
public:
    InvalidArgumentException( ir.t arg, char* pFile,
                              int nLine)
     :Exception("Неверный аргумент", pFile, nLine)
     {
         invArg = arg;
     }
    virtual void display(ostream& out)
     {
         Exception::display(out);
         cut << "Аргумент равен " << invArg << endl;
     }
protected:
    int invArg;
J;
```

Несмотря на то что в блоке catch указан класс Exception, исключение InvalidArgumentException будет успешно обработано, поскольку InvalidArgumentException ЯВЛЯЕТСЯ Exception, а функция-член display!) полиморфна.

Глава 28

Множественное наследование

В этой главе...

- ✓ Механизм множественного наследования
- ✓ Устранение неоднозначностей множественного наследования
- ✓ Виртуальное наследование
- ✓ Конструирование объектов
- ✓ Отрицательные стороны множественного наследования

В иерархиях классов, которые рассматривались в этой книге, каждый класс наследовался от одного прародителя. Такое одиночное наследование подходит для описания большинства объектов реального мира. Однако некоторые классы представляют собой сочетание нескольких классов в одном.

Примером такого класса может служить диван-кровать. Как видно из названия, это и диван и кровать (правда, кровать не очень удобная). Таким образом, этот предмет интерьера наследует свойства как дивана, так и кровати. В терминах С++ эту ситуацию можно описать следующим образом: класс может быть наследником более чем одного базового класса. Такое наследование называется *множественным.*

Механизм множественного наследования

Чтобы увидеть множественное наследование в действии, я продолжу пример с диваном-кроватью. На рис. 28.1 приведена схема наследования дивана-кровати (класс `SleeperSofa`). Обратите внимание, что этот класс наследует свойства и от класса Bed (Кровать), и от класса Sofa (Диван), т.е. наследует свойства обоих классов.

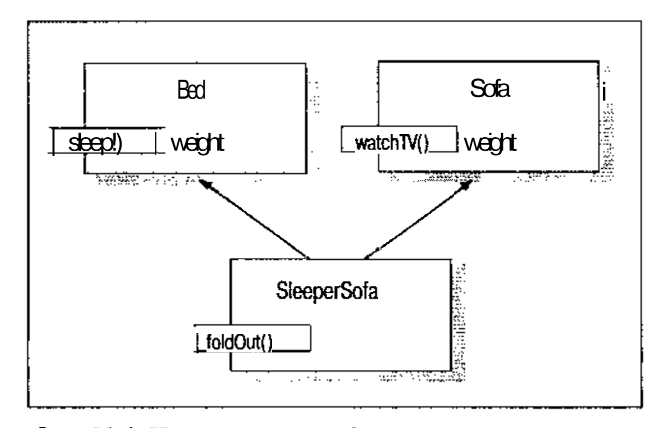


Рис. 28.1. Иерархия классов дивана-кровати

Программная реализация класса SleeperSofa выглядит следующим образом:

```
class Bed
{
public:
    Bed(){}
    void sleep(){}
    int weight;
};

class Sofa
{
public:
    Sofa(){}
    void watchTV(){}
    int weight;
};

// SleeperSofa — это одновременно и кровать и диван
class SleeperSofa : public Bed, public Sofa
{
public:
    SleeperSofa(){}
    void foldOut(){}
};

int main(int argcs, char* pArgs[])
{
    SleeperSofa ss;

    // Вы можете смотреть телевизор, сидя на диване...
    ss.watchTV();      // Sofa::watchTV()

    //...а потом можете разложить его...
    ss.foldOut();      // SieeperSofa::foldOut()

    //...и спать на нем как на кровати
    ss.sleep();        // Bed::sleep()
    return 0;
}
```

В этом примере класс SieeperSofa наследует оба класса — Bed и Sofa. Это видно из их наличия в объявлении класса SieeperSofa, который наследует все члены от обоих базовых классов. Таким образом, допустимы оба вызова — как ss.sleep(), так и ss.watchTV(). Вы можете использовать SieeperSofa и как Bed, и как Sofa. Кроме того, класс SieeperSofa имеет собственные члены, например foldOut().

Устранение неоднозначностей множественного наследования

Будучи весьма мошной возможностью языка, множественное наследование может стать в то же время и источником проблем. Одну из них можно увидеть уже в предыдущем примере. Обратите внимание, что оба класса — Bed и Sofa — содержат член weight (вес). Это логично, потому что они оба имеют некоторый вполне измеримый вес. Вопрос: какой именно член weight наследует класс SieeperSofa?

Ответ прост: оба. Класс sleeperSofa наследует отдельный член Bed::weight и отдельный член Sofa: :weight. Поскольку они оба имеют одно и то хе имя. обращения к weight теперь являются двузначными, если только не указывать явно, к какому именно weight мы намерены обратиться. Это демонстрирует следующий фрагмент кода:

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    cout « "Зес = "
         << ss.weight  //неправильно — какой именно вес?
         << "\п";
```

Теперь в программе нужно явно указывать, какая именно переменная weight нужна, используя для этого имя базового класса. Приведенный ниже пример вполне корректен.

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    cout << "Вес дивана = "
         << ss.Sofa::weight  //укажем, какой именно вес
         << "\n";
}
```

Хотя такое решение и устраняет ошибку, указание имени базового класса во внешнем приложении нежелательно: ведь при этом информация о внутреннем устройстве класса должна присутствовать за ею пределами. В нашем примере функция fn () должна обладать сведениями о том, что класс SleeperSofa наследуется от класса Sofa. Такие конфликты имен невозможны при одиночном наследовании, но служат постоянным источником неприятностей при наследовании множественном.

Виртуальное наследование

В случае класса SleeperSofa конфликт имен weight является, по сути, небольшим недоразумением. Ведь на самом деле диван-кровать не имеет отдельного веса как кровать, и отдельного веса как диван. Конфликт возник потому, что такая иерархия классов не вполне адекватно описывает реальный мир. Дело в том, что разложение на классы оказалось неполным.

Если немного подумать нал этой проблемой, становится ясно, что и кровать и диван являются частными случаями некоторой более фундаментальной концепции мебели (думаю, можно было предложить нечто еше более фундаментальное, но для нас достаточно ограничиться мебелью). Вес является свойством любой мебели, что показано на рис. 28.2.

Отделение класса Furniture (мебель) должно устранить конфликт имен. Итак, с чувством глубокого удовлетворения и облегчения, в предвкушении успеха я реализую новую иерархию классов:

```
#include <iostream.h>
//Furniture — более фундаментальная концепция; этот класс
              имеет свойство "weight" — "вес"
class Furniture
{
public:
```

```
    Furniture() {}
    int weight;
};

class Bed : public Furniture
{
public:
    Bed() {}
    void sleep() {}
};

class Sofa : public Furniture
{
public:
    Sofa(){}
    void watchTV(){}
};

class SleeperSofa : public Bed, public Sofa
{
public:
    SleeperSofa() {}
    void foldOut() {}
};

void fn()
{
    SleeperSofa ss;
    cout << "вес = "
         << ss.weight
         << "\n";
}
```

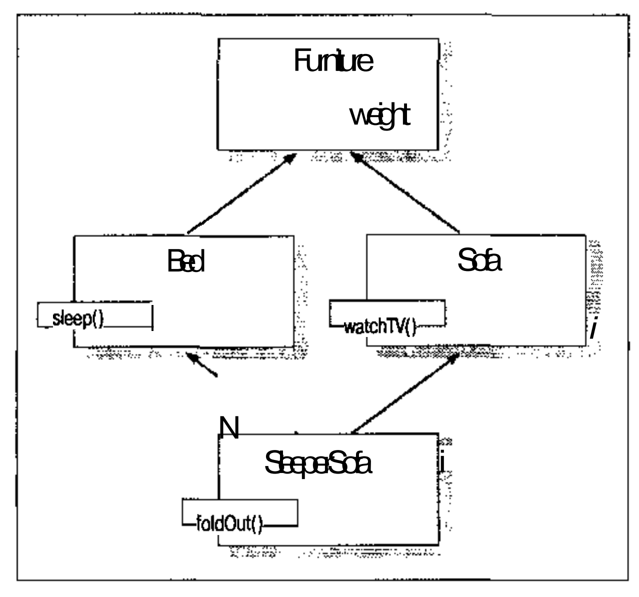


Рис. 28.2. Выделение общих свойств кровати и дивана

М-да... "Не говори "гоп", пока не переехал Чоп" — новая иерархия классов совершенно нас не спасает. weight остается неоднозначным. Попробуем привести ss к классу Furniture?

```
#include <iostream.h>

void fn()
{
    SleeperSofa ss;
    Furniture* pF;
    pF = (Furniture*)&ss;
    cout « "Вес = "
         << pF -> weight
         <<"\n";
};
```

Приведение ss к классу Furniture тоже ничего не дает. Более того, я получил какое-то подозрительное сообщение о том, что приведение SleeperSofa* к классу Furniture* неоднозначно. Да что, в конце концов, творится?

На самом деле все довольно просто. Класс SleeperSofa не наследуется напрямую от класса Furniture. Сначала Furniture наследуют классы Bed и Sofa, а уж потом SleeperSofa наследуется от этих классов. Класс SleeperSofa выглядит в памяти так, как показано на рис. 28.3.

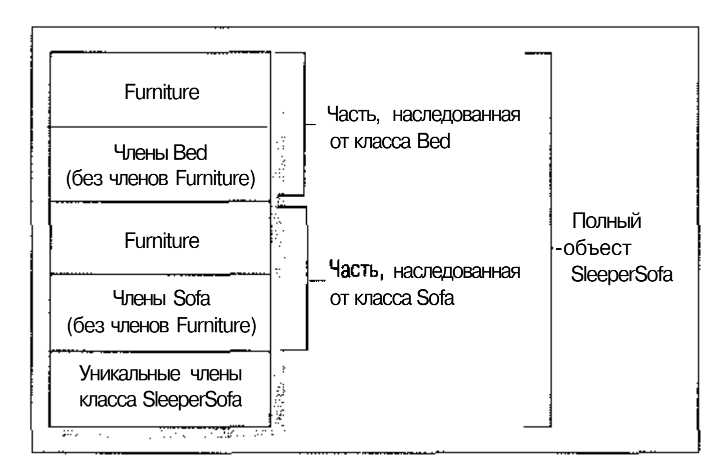


Рис. 28.3. Расположение класса SleeperSofa в памяти

Как видите, SleeperSofa состоит из класса Bed, за которым в полном составе следует класс Sofa, а после него — уникальные члены класса SleeperSofa. Каждый из подобъектов класса SleeperSofa имеет свою собственную часть Furniture, поскольку они оба наследуются от этого класса. В результате объекты класса SleeperSofa содержат два объекта класса Furniture.

Таким образом, становится ясно, что я не сумел создать иерархию, показанную на рис. 28.2. Иерархия наследования, которая была создана в результате выполнения предыдущей программы, показана на рис. 28.4.

SleeperSofa содержит два объекта класса Furniture, что является явной бессмыслицей! Я хочу, чтобы SleeperSofa наследовал только одну копию Furniture и чтобы Bed и Sofa имели к ней доступ. В С++ это достигается *виртуальным наследованием,* поскольку в этом случае используется ключевое слово virtual.

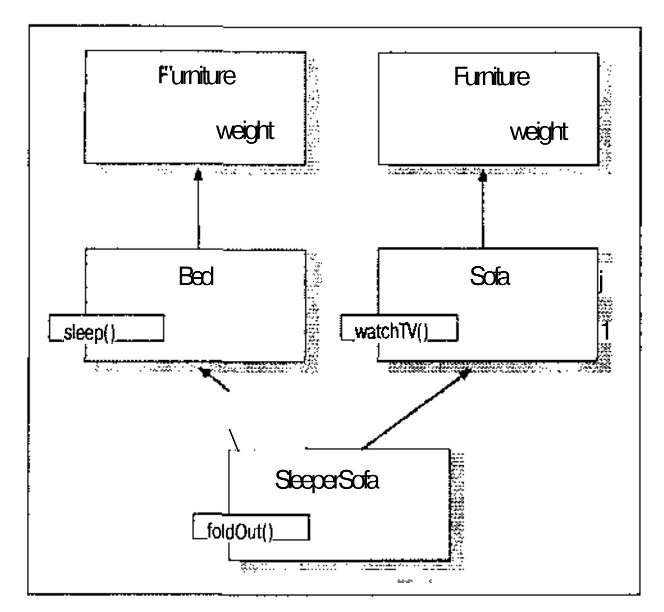


Рис. 28.4. Результат попытки создания иерархии классов



Я ненавижу перегрузку терминов (что произошло в данном случае) — ведь виртуальное наследование не имеет ничего общего с виртуальными функциями!

Вооруженный новыми знаниями, я возвращаюсь к классу SleeperSofa и реализую его так, как показано ниже.

```
#include <iostream.h>
class Furniture
{
public:
    Furniture() {}
    int weight;
};

class Bed : virtual public Furniture
{
public:
    Bed() {}
    void sleep(){}
};

class Sofa : virtual public Furniture
{
public:
    Sofa() {}
    void watchTV(){}
};

class SleeperSofa : public Bed, public Sofa
{
public:
    SleeperSofa() : Sofa(), Bed() {}
    void foldOut(){}
```

```
};

void fn()
{
    SleeperSofa ss;
    cout << "Вес = "
         << ss.weight
         << "\n";
}
```

Обратите внимание на ключевое слово virtual, использующееся при наследовании классов Bed и Sofa от класса Furniture. Оно означает примерно следующее: "Дайте-ка мне копию Furniture, но если она уже существует, то я использую именно ее". В итоге класс SleeperSofa будет выглядеть, как показанно на рис. 28.5.

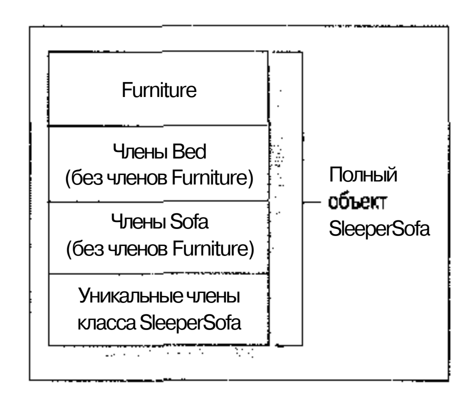


Рис. 28.5. Расположение класса SleeperSofa в памяти при использованиивиртуальногонаследования

Из этого рисунка видно, что класс SleeperSofa содержит Furniture, а также части классов Bed и Sofa, не содержащие Furniture. Далее находятся уникальные для класса SleeperSofa члены (элементы в памяти не обязательно будут располагаться именно в таком порядке, но в данном обсуждении это несущественно).

Теперь обращение к члену weight в функции fn() не многозначно, поскольку SleeperSofa содержит только одну копию Furniture. Наследуя этот класс виртуально, мы получили желаемую структуру наследования (см. рис. 28.2).

Если виртуальное наследование так хорошо решает проблему неоднозначности, почему оно не является нормой? Во-первых, потому, что виртуально наследуемый класс обрабатывается иначе, чем обычный наследуемый базовый класс, что, в частности, выражается в повышенных накладных расходах. Во-вторых, у вас может появиться желание иметь две копии базового класса (хотя это случается весьма редко). Вспомним наши старые упражнения со студентами и преподавателями и допустим, что TeacherAssistant (помощник преподавателя) является одновременно и Teacher (преподавателем) и Student (студентом), которые, в свою очередь, являются подклассами Academician. Если университет даст помощнику преподавателя два идентификатора — и студента и преподавателя, то классу TeacherAssistant понадобятся две копии класса Academician.

Конструирование объектов

При конструировании объектов с использованием множественного наследования должен выполняться ряд правил.

1. Сначала вызываются конструкторы для каждого виртуального базового класса в порядке наследования.
2. Затем вызываются конструкторы каждого невиртуального базового класса в порядке наследования.
3. После этого вызываются конструкторы всех объектов-членов класса в том порядке, в котором эти объекты-члены объявлены в классе.
4. И наконец, вызывается конструктор самого класса.

Обратите внимание, что базовые классы конструируются в порядке наследования, а не в порядке расположения в строке конструктора.

Отрицательные стороны множественного наследования

Должен признаться, что не все, кто работает с объектно-ориентированным программированием, считают механизм множественного наследования удачным. Кроме того, многие объектно-ориентированные языки вообще не поддерживают множественного наследования, реализация которого, кстати, далеко не самая простая вещь. Конечно, множественное наследование — это проблема компилятора (вернее, того, кто пишет компилятор). Однако оно требует больших накладных расходов по сравнению с программой с обычным наследованием, а эти накладные расходы становятся уже проблемой программиста.

Не менее важно и то, что множественное наследование открывает путь к дополнительным ошибкам. Во-первых, неоднозначность, подобная описанной в предыдущем разделе, может превратиться в большую проблему. Во-вторых, при наличии множественного наследования преобразования указателя на подкласс в указатель на базовый класс иногда приводят к запутанным и непонятным изменениям этого указателя. Все эти тонкости я оставляю на совести разработчиков языка и компилятора, но хочу продемонстрировать программу, которая из-за этого может привести к непредсказуемым результатам:

```
#include <iostream.h>

class Base1 {int mem;};
class Base2 (int mem;);
class Subclass : public Basel, public Base2 {};

void fn(SubClass* pSC)
{
    Basel* pB1 = (Base1*)pSC;
    Base2 * pB2 = (Base2*)pSC;
    if((void*)pB1 == (void*)pB2)
    {
        cout <<' "Указатели численно равны\n" ;
    }
}

int main(int args, char* pArgs[])
{
    Subclass sc;
    fn(&sc);
    return 0;
}
```

Указатели pB1 и pB2 не равны, несмотря на то что они указывают, по сути, на один объект *psc.

Думаю, вам стоит избегать множественного наследования, пока вы в полной мере не освоите C++. Обычное наследование тоже достаточно мощный механизм. Исключением может стать библиотека Microsoft Foundation Classes (MFC), в которой множественное наследование используется сплошь и рядом. Однако эти классы тщательно выверены профессиональными высококвалифицированными программистами.

Только не поймите меня неправильно! Я не против множественного наследования. То, что Microsoft и другие компании эффективно используют множественное наследование в своих классах, доказывает, что так можно делать. Если бы этот механизм не стоил того, они бы его не использовали. Однако это отнюдь не значит, что множественное наследование имеет смысл применять, едва познакомившись с ним.

Часть VI

Великолепная десятка



В этой части...

Ни одна книга ...для "чайников" *не может обойтись без "Великолепных десяток".*
В главе 29, ***"Десять** способов избежать ошибок",*
описано десять способов, благодаря которым вы сможете оградить свою программу от ошибок.
Многое из того, что здесь сказано, без особых оговорок подходит и для С.

Глава 29

Десять способов избежать ошибок

В этой главе...

- ✓ Включение всех предупреждений и сообщений об ошибках
- ✓ Добейтесь чистой компиляции
- ✓ Используйте последовательный стиль программирования
- ✓ Ограничивайте видимость
- ✓ Комментируйте свою программу
- ✓ Хотя бы один раз выполните программу пошагово
- ✓ Избегайте перегрузки операторов
- ✓ Работа с кучей
- ✓ Используйте исключительные ситуации для обработки ошибок
- ✓ Избегайте множественного наследования

Включение всех предупреждений и сообщений об ошибках

Синтаксис C++ позволяет проверять все и вся. Когда компилятор встречается с конструкцией, которую он не может понять, у него не остается никакого выбора, кроме генерации сообщения об ошибке. И хотя компилятор честно пытается перейти к следующей строке программы, он даже не сочтет нужным создать выполняемую программу.

Выключение сообщений об ошибках и предупреждений подобно отключению красных габаритных огней на вашей машине, потому что они вас раздражают. Игнорирование проблемы не заставит ее исчезнуть. Если ваш компилятор имеет режим абсолютной проверки кода, включите его. И Microsoft и Borland предоставляют режим "Включить все сообщения" (Enable All Messages), который должен постоянно находиться в рабочем состоянии. В конце концов эта многословность сбережет ваше время.

Рассматривая ваш исходный код, умный компилятор C++ помимо ошибок ищет всякие подозрительные конструкции. Вот пример:

```
#include "student.h"
#include "class.h"
Student* addNewStudent (Class class, char *pName,
                        SSNumber ss)
{
    Student pS;
    if(pName != 0)
    {
        pS = new Student(pName, ss);
        class.addStudent(pS);
    }
    return pS;
}
```

Умный компилятор (типа Visual C++) понимает, что если pName равно 0, то переменная pS никогда не будет проинициализирована, и выводит сообщение об этом и обращает ваше внимание, что было бы неплохо взглянуть на эту проблему еще раз.

Добейтесь чистой компиляции

Не начинайте отладку кода, пока не избавитесь от всех предупреждений (или хотя бы не поймете их причину). Выключение режима вывода всех сообщений об ошибках, чтобы они не надоедали, не приведет ни к чему хорошему. Если вы не поняли сообщение сразу, посидите и разберитесь с ним. Проблемы появляются именно там, где вы чего-то не понимаете.

Используйте последовательный стиль программирования

Программирование с использованием чистого последовательного стиля не только улучшает читаемость программы, но и уменьшает количество ошибок программирования. Помните, что, чем меньше сил будет уходить на расшифровку кода, тем больше их останется на обдумывание логики создаваемой программы. Хороший стиль программирования позволяет легко справляться с такими задачами:

- ✓ различать имена классов, имена объектов и имена функций;
- ✓ получать определенную информацию об объекте по его имени;
- ✓ отличать команды препроцессора от команд C++;
- ✓ различать блоки программы C++ по уровню отступов.

Кроме того, вы должны приучиться создавать стандартный заголовок модулей. В таких заголовках следует приводить информацию о функциях или классах, содержащихся в данном модуле, об имени автора (имеется в виду ваше имя), о дате создания и версии используемого компилятора, а также об истории исправлений и модификаций.

Очень важно, чтобы программисты, работающие над одним проектом, имели одинаковый стиль программирования. Попытка разобраться в программе, состоящей из фрагментов с разными стилями программирования, довольно тоскливое занятие.

Ограничивайте видимость

Ограничение видимости внутреннего содержимого класса для внешнего мира — один из краеугольных камней объектно-ориентированного программирования. Класс сам отвечает за свое содержимое, а приложение отвечает лишь за использование класса для решения поставленных задач.

В частности, ограниченная видимость означает, что данные-члены не должны быть доступны извне класса, т.е. они должны быть защищенными или закрытыми. Кроме того, должны быть защищены функции-члены, о которых внешние приложения не должны знать.

Открытые функции-члены должны как можно меньше доверять внешнему коду — любой аргумент, переданный открытой функции-члену, должен рассматриваться как по-

тенциальный источник ошибки, пока не будет достоверно доказано обратное. Функция, приведенная ниже, представляет собой мину, которая только и ждет своего часа.

```
class Array
{
public:
    Array(int s)
    {
        size=0;
        pData = new int[s];
        if (pData)
        {
            size = s;
        }
    }
    ~Array()
    {
        delete pData;
        size = 0;
        pData = 0;
    }
    //Вернуть или установить данные в массиве
    int data fint index)
    {
        return pData[index];
    }
    int data (int index, int newValue)
    {
        int oldValue = pData[index];
        pData[index] = newValue;
        return oldValue;
    }
protected:
    int size;
    int *pData;
};
```

Функция data(int) позволяет внешнему приложению читать данные из Array. Эта функция слишком доверчива: она не проверяет, находится ли переменная index в допустимом диапазоне. А что, если она выйдет за предусмотренные границы? Функция data(int, int) с этой точки зрения еще хуже, поскольку производит запись в неизвестное место в памяти.

В следующем примере показано, как осуществить такую проверку. Для краткости приведен только пример функции data(int).

```
int data(unsigned int index)
{
    if(index >= size)
    {
        cout << "Индекс массива вне "
                "допустимого диапазона ("
             « index « "}\n";
        return 0;
    }
    return pData[index];
}
```

Теперь, если переменная index будет выходить за пределы допустимого диапазона, это будет выявлено на этапе проверки (беззнаковость переменной index избавляет от необходимости проверки на отрицательность значения индекса).

Комментируйте свою программу

Я считаю, что можно избежать излишних ошибок, если комментировать программу в процессе ее написания, вместо того чтобы ждать, когда программа заработает, и только потом возвращаться и добавлять комментарии к тексту.

Понятно, что никому не хочется тратить время на написание объемистых заголовков и пояснений к функциям, но я думаю, вы всегда найдете пару минут, чтобы добавить краткий комментарий к программе. Когда вы вернетесь к коду, написанному недели две назад, короткие и содержательные комментарии значительно облегчат "вживание" в программу.

Кроме того, последовательно используемые отступы и соглашения об именах сделают код гораздо доступнее для чтения и понимания. Конечно, очень приятно, когда код легко читать по завершении написания программы, но просто необходимо, чтобы код хорошо воспринимался и при написании — именно тогда, когда вам действительно нужна помощь.

Хотя бы один раз выполните программу пошагово

Вам, как программисту, очень важно понимать, как работает ваша программа. Ничто не может дать облегчить понимание, чем пошаговое выполнение программы с помощью хорошего отладчика (обычно для этого вполне подходят отладчики, включенные в среды разработки компиляторов).

Когда некоторая функция готова и может быть добавлена к программе, ее следует тщательно проверить хотя бы один раз, проходя все возможные варианты ветвления потока управления. Гораздо легче отлавливать ошибки в отдельной функции, чем в комплексе с другими функциями программы.

Избегайте перегрузки операторов

Настоятельно советую избегать перегрузки операторов, за исключением операторов ввода-вывода `operator<<()` и `operator>>()` и оператора присвоения `operator=()`, пока вы как следует не освоитесь в C++. Хотя хороший набор перегруженных операторов и может значительно повысить полезность класса и читаемость кода, перегрузка почти никогда не бывает крайне необходимой и может значительно усложнить жизнь начинающего программиста.

После того как вы интенсивно поработаете с C++ по меньшей мере несколько месяцев, можете начинать перегружать операторы как вам заблагорассудится.

Работа с кучей

Основное правило, касающееся кучи, заключается в том, что выделение и освобождение памяти из кучи должно происходить на одном уровне. Если функция-член `MyClass::create()` выделяет блок памяти и возвращает его вызывавшему коду, то должна существовать и функция `MyClass::release()`, которая освобождает блок памяти, возвращая его в кучу. `MyClass::create()` не должна требовать от вызывающей функции самостоятельного освобождения памяти. Это, конечно, не

помогает избежать всех проблем (например, вызывающая функция может просто "забыть" вызвать MyClass::release()), однако все же снижает вероятность их возникновения.

Используйте исключительные ситуации для обработки ошибок

Механизм исключений введен в C++ специально для удобства и эффективности обработки ошибок. Теперь, когда эта возможность стандартизована, вы можете спокойно использовать ее.

Избегайте множественного наследования

Множественное наследование, как и перегрузка операторов, на порядок усложняют программу, что совсем не нужно начинающему программисту. К счастью, большинство отношений реального мира могут быть выражены с помощью одиночного наследования (некоторые утверждают, что множественное наследование вообще не нужно, но я не из таких).

В любом случае вы можете спокойно использовать классы с множественным наследованием из коммерческих библиотек, например классы Microsoft MFC.

Лишь убедившись в том, что вы твердо понимаете концепции C++, можно начинать экспериментировать с множественным наследованием. Вы будете уже достаточно подготовлены к неожиданным ситуациям, которые могут возникнуть при использовании этого механизма.

Приложение А

Словарь терминов

this	this	Указатель на текущий объект. this — это скрытый первый аргумент всех нестатических функций-членов, Всегда имеет тип "указатель на текущий класс".
v_table	v_table	Таблица, которая содержит адреса виртуальных функций класса, Каждый класс, который имеет одну или больше виртуальных функций, обязательно содержит такую таблицу.
Абстрактный класс	Abstract class	Класс, который содержит одну или несколько чисто виртуальных функций, Создание объекта такого класса невозможно.
Абстракция	Abstraction	Концепция упрощения реального мира, который разбивается на основные элементы. Абстракция позволяет классам отображать реальный мир. Без абстракции классы представляли бы реальность а ее безнадежно запутанном виде.
Базовый класс	Base class	Класс, из которого порождаются другие классы.
Виртуальная функция-член	Virtual member function	Функция-член, которая вызывается полиморфно. См. *полиморфизм.*
Внешняя (outline) функция	Outline function	Обычная функция, тело которой находится, как правило, в месте ее объявления. Все последующие обращения к функции приводят к ее вызову из одного места в памяти. См. также *встраиваемаяфункция.*
Встраиваемая (inline) функция	Inline function	Функция, подставляемая в точке вызова; очень похожа на макроопределение.
Выражение	Expression	Последовательность подвыражений и операторов. В языках С и С++ выражение всегда имеет тип и значение.
Глобальная переменная	Global variable	Переменная, объявленная вне каких-либо функций и благодаря этому доступная для всех функций.
Глубокое копирование	Deep copy	Когирование объекта, состоящее в реплицировании содержимого объекта и ресурсов, которыми владеет объект, включая объекты, на которые указывают данные-члены копируемого объекта.
Друг	Friend	Функция или класс, которые не являются членом данного класса, однако имеют доступ к защищенным членам класса.
Закрытый	Private	Член класса, доступный только для других членов того же класса.
Защищенный	Protected	Член класса, доступный только для членов этого класса и его подклассов. Защищенные члены класса не являются общедоступными.
Классификация	Classification	Объединение похожих объектов. Например, теплокровные, живородящие, вскормленные молоком матери существа, объединяются в один класс — *млекопитающие.*
Конструктор	Constructor	Специальная функция-член, которая автоматически вызывается в момент создания объекта.

Конструктор по умолчанию	Default constructor	Конструктор, который содержит пустой список аргументов.
Контрольное поле	Signature field	Нестатический член, которому присваивается определенное значение. Это значение может быть затем проверено функциями-членами для контроля корректности указателя на объект. Это очень эффективный отладочный прием.
Копирующий конструктор (конструктор копирования)	Copy constructor	Конструктор, аргументом которого является ссылка на объект того же класса. Например, копирующий конструктор для класса z объявляется как `Z::z(z&)`.
Куча	Heap	Память, выделяемая для программы с помощью функции `malloc()`. Эта память должна быть освобождена посредством функции `free()`.
Метод	Method	См. *функция-член.*
Наследование	Inheritance	Способность классов C++ перенимать свойства уже существующих классов.
Открытый	Public	Член класса, доступный извне этого класса.
Объектно-ориентированное программирование	Object-oriented programming	Программирование, которое базируется на принципах сокрытия данных, абстракции и полиморфизме.
Объявление прототипа функции	Function prototype declaration	Объявление функции, которое не содержит кода (тела функции).
Объявление функции	Function declaration	Описание функции, состоящее в указании имени функции, имени класса, с которым связана эта функция (если связана), количества и типа всех аргументов, а также типа возвращаемого функцией значения,
Функция обратного вызова	Callback function	Функция, вызываемая операционной системой в ответ на определенное событие.
Парадигма	Paradigm	В области программирования — способ мышления. Используется в контексте, например: объектно-ориентированная парадигма и функциональная парадигма.
Перегрузка	Overloading	Предоставление двум разным функциям одинакового имени. Такие функции различаются компилятором по количеству и типам их аргументов.
Перегрузка операторов	Operator overloading	Определение функциональности встроенных операторов при использовании с пользовательскими типами.
Переопределение	Overriding	Создание в подклассе функции с тем же именем и аргументами, что и в базовом классе. См. *полиморфизм* и *виртуальная функция-член.*
Поверхностное копирование	Shallow copy	Побитовое копирование.
Подкласс	Subclass	Класс, наследующий открытые свойства другого класса. Если `Undergraduate` — подкласс класса `Student`, Значит, `Undergraduate` ЯВЛЯЕТСЯ `Student`.
Позднее связывание	Late binding	Процесс, который используется в C++ для осуществления полиморфизма.

Полиморфизм	Polymorphism	Способность решать, какая из перегруженных функций-членов должна быть вызвана в зависимости от текущего типа времени исполнения объекта.
Порожденный класс	Derived class	Класс, который наследует другой класс.
Поток ввода-вывода	Stream i/o	Ввод-вывод в C++, основанный на перегруженных операторах operator << и operator >>. Прототипы этих функций находятся во включаемом файле lostream.h.
Раннее связывание	Early binding	Нормальный (не полиморфный) метод вызова. Все вызовы в С осуществляются с использованием раннего связывания.
Расширяемость	Extensibility	Способность добавления новых возможностей в класс без изменения существующего кода, который используется этим классом,
Сегмент данных	Data segment	Блок памяти, в котором С и C++ хранят глобальные и статические переменные, См. также *сегмент кода* и *сегмент стека.*
Сегмент кода	Code segment	Часть программы, которая содержит исполняемые инструкции.
Сегмент стека	Stack segment	Часть программы в памяти, которая содержит нестатические локальные переменные.
Сигнатура функции	Function signature	То же самое, что и полное имя функции (включающее типы аргументов).
Сокращенное вычисление	Short-circuit evaluation	Метод, при котором правая часть подвыражения не вычисляется в том случае, если это не изменит результат выражения. Такая ситуация может возникнуть с двумя операторами: ¦! и &s, Например, если в выражении a &&ь левый аргумент равен o (false), то нет необходимости вычислять правый аргумент, поскольку результат все равно будет равен o.
Ссылочная переменная	Reference variable	Переменная, которая выступает в качестве псевдонима другой переменной.
Статическая функция-член	Static member function	Функция-член, не имеющая указателя this.
Статический член	Static data member	Член, который не связан с отдельными экземплярами класса, Для каждого класса существует только один экземпляр каждого статического члена, независимо от того, сколько объектов этого класса было создано.
Тип переменной	Variable type	Определяет размер и внутреннюю структуру переменной. Примерами встроенных типов являются int. char, double и float.
Указатель	Pointer variable	Переменная, которая содержит адрес.
Устранение неоднозначности	Disambiguation	Процесс выбора того, к какой из перегружаемых функций относится вызов а зависимости от прототипов перегруженных функций.
Фаза анализа	Analysis phase	Одна из фаз создания приложения, во время которой анализируется поставленная задача и ее основные элементы.
Фаза кодирования	Coding phase	Фаза, во время которой результаты фазы разработки воплощаются в коде.
Фаза разработки	Design phase	Фаза создания приложения, во время которой формулируется решение проблемы. Входными данными для нее является результат фазы анализа.

Функция-член	Member function	Функция, определенная как часть класса; объявляется так же, как и данные-члены.
Чисто виртуальная функция	Pure virtual function	Виртуальная функция, которая не имеет тела.
Член класса	Class member	См. *статический член.*
Член экземпляра	Instance member	Еще один синоним обычного (не статического) члена.
Экземпляр класса	Instance of a class	Объект объявленного типа. Например, i является экземпляром класса int в объявлении int i.
ЯВЛЯЕТСЯ	IS_A	Тип взаимосвязи между подклассом и классом. Например, Mallard ЯВЛЯЕТСЯ Duck; это означает, что объект класса Mallard является также объектом класса Duck.

Предметный указатель

З

И

К

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Э

Научшыюпущтое издание

Стефан Р. Дэвис

С++ для "чайников", 4-е издание

В издании использованы карикатуры американского художника Рича Теннанта

Литературный редактор *Т. //. Капгородова*
Верстка *В.II. Бордюк*
Художественный редактор *ВТ. Пав.иотин*
Технический редактор *ГЛ. Горобец*
Корректоры *Л.А. Гордиенко, О. В. Мишутина, Л. В. Чернокозинская*

Издательский дом "Вильяме".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 28.02.2003. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 27,3. Уч.-изд. л. 15,83.
Доп. тираж 4000 экз. Заказ № 2507.

Отпечатано с диапозитивов вФГУП "Печатный двор"
Министерства РФ по делам печати.
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр.. 15.